Combining Data Driven Programming with Component Based Software Development

With applications in Geovisualisation and Dynamic Data Driven Application Systems

ANTHONY ANDREW JONES

Doctor of Philosophy

ASTON UNIVERSITY

January 2008

Thesis Summary

Software development methodologies are becoming increasingly abstract, progressing from low level assembly and implementation languages such as C and Ada, to component based approaches that can be used to assemble applications using technologies such as JavaBeans and the .NET framework. Meanwhile, model driven approaches emphasise the role of higher level models and notations, and embody a process of automatically deriving lower level representations and concrete software implementations.

The relationship between data and software is also evolving. Modern data formats are becoming increasingly standardised, open and empowered in order to support a growing need to share data in both academia and industry. Many contemporary data formats, most notably those based on XML, are self-describing, able to specify valid data structure and content, and can also describe data manipulations and transformations. Furthermore, while applications of the past have made extensive use of data, the runtime behaviour of future applications may be *driven by data*, as demonstrated by the field of dynamic data driven application systems.

The combination of empowered data formats and high level software development methodologies forms the basis of modern game development technologies, which drive software capabilities and runtime behaviour using empowered data formats describing game content. While low level libraries provide optimised runtime execution, content data is used to drive a wide variety of interactive and immersive experiences.

This thesis describes the Fluid project, which combines component based software development and game development technologies in order to define novel component technologies for the description of data driven component based applications. The thesis makes explicit contributions to the fields of component based software development and visualisation of spatiotemporal scenes, and also describes potential implications for game development technologies. The thesis also proposes a number of developments in dynamic data driven application systems in order to further empower the role of data in this field.

Keywords: Components, Data Driven, Visualisation

To my grandparents:

for all the happy memories.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dan Cornford, whose guidance, friendship and support have been invaluable during the course of my PhD. Thank you for being both a mentor and a friend, for encouraging me to explore and learn, and for making this journey of discovery and development so enjoyable and rewarding.

I am also very grateful for a variety of input from Michal Konečný. Your comments have often made me look a little closer at what I'm doing, and my research has certainly benefited from the additional scrutiny! Many thanks for your helpful support.

Jan Duracz and Chris Mantle have been a great source of insight, ideas, and interesting diversions throughout the past few years. I would like to say thank you to Jan: for our many colourful discussions, for your interest and support when the work was hard, for the fun and various distractions when the work was easy, and for trying to teach me lots of interesting stuff.

I would also like to thank Chris for his interest in, and contributions to, the computer graphics side of my work. Thank you for opting to work with me for two years running, for improving upon my render system prototype in order to build the PirateHat rendering library, and for your contributions to the subsequent Eurographics publication.

I owe a big thank you to the game developers who have contributed to my work: I am indebted to Scott Bilas and Eric Malafeew for taking the time to provide me with some valuable input.

Finally, I would like to offer my heartfelt thanks to Helen Eley, whose love and support over the past three years have made everything much, much easier. Thank you for pushing me to do my best, for making every day a happier one, and for always inspiring me to be a better person.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software development methodologies are evolving, incorporating increasingly abstract concepts, representations and manipulations. Early assembly languages had a one to one correspondence with the binary machine instructions understood by computer hardware platforms. Due to their fundamental nature, and the need for programmers to perform a variety of low level tasks, larger programs were difficult to write and maintain using assembly languages. Several generations of higher level languages have since introduced a number of increasingly abstract notations and concepts, with compilers translating software programs to their corresponding binary representations.

Throughout the evolution of software programming languages, many concepts and abstractions have allowed programmers to describe data structures and algorithms more productively. For example, early languages introduced a range of data representations including numbers and textual strings, and enforced their correct use throughout program execution so that programmers could rely on automated enforcement of type safety. Later languages supporting procedures allowed programmers to encapsulate and subsequently reuse the implementation of algorithms, while automating the management of control flow through procedural execution. More recently, the object oriented programming paradigm has allowed programmers to conceptualise software applications using abstractions corresponding to the real world, and to separate their implementation into independent units of behaviour with well defined interfaces.

Component based approaches continue the evolution of software development methodologies into higher levels of abstraction by building upon concepts established by the object oriented paradigm. Component based methodologies draw on a range of analogies such as computer hardware and Lego, and focus on plugging together independent software components in order to form component software applications. In practice, component based methods make use of binary (compiled) representations providing computational functionality, which are then assembled into applications by interconnecting their interfaces. The process of component selection and interconnection is driven by high level notations specifying *component topologies*, where component topology refers to the connectivity of components providing computational encapsulations. While a range of composition languages define component based applications as software structures, Architecture Description Languages (ADLs) extend the component based approach to the architectural level of abstraction while also considering non-functional factors such as composition correctness and the quality of service of the described application.

Component based methodologies explicitly separate computational elements from their topology, aiming to produce independently developed software components that may ultimately be obtained and assembled by third parties in order to form software applications. Although a high level notation may be used to specify component topology, the components themselves are commonly written using lower level languages. While component based methodologies allow the bottom-up definition of software, model driven approaches adopt a top-down approach to software development. However, model driven methodologies are far more ambitious in that they aim to derive complete software applications from high level abstract models, by incrementally generating lower level representations until implementation details may be obtained.

While the eventual aim of research into model driven software development is the complete and automatic generation of software implementation, current solutions typically rely on human intervention or the provision of concrete code using components or lower level representations. However, if such research is successful, higher level notations could become programming languages of the future, and programmers as we currently know them may become obsolete.

While software development methodologies continue to incorporate incremental abstractions and higher level concepts, data is becoming increasingly important to a wide range of scientific disciplines. A growing number of scientists are turning to automated processes and technologies in order to collect, store, analyse and interpret experimental data [1]. Meanwhile, such experiments are producing increasing volumes of data, with petabyte data sets being predicted for the near future [2].

As computers become more ubiquitous and technologies such as the Internet become more readily available, the scientific community have also begun to make widespread use of data sharing. Purchasing data may be more attractive when the cost of measuring or producing results is prohibitive. Meanwhile, research efforts spanning different disciplines, countries and continents are able to communicate their data more effectively. Many private companies and public organisations are also making a wide variety of data available via their websites.

The communication of data may further benefit from recent changes in data representations. In many areas, closed proprietary binary formats have been replaced with open, standardised and text-based formats such as XML. As binary data consists of a sequence of ones and zeros, meaningful analysis requires the use of a computer. On the other hand, text-based formats may be read and interpreted by humans, although the use of tools is recommended for all but the most simple of data files. Data may also be accompanied by metadata, describing attributes of the data, providing information such as the data's owner, origin, currency, unit of measurement, what measurement process was used to obtain the data, and more recently, the reliability, accuracy and uncertainty of the data.

Today's data availability, communication and processing capabilities are allowing scientists to tackle a wide variety of problems with increasing scale, complexity and diversity. While the availability of more computationally demanding software applications has increased and multi-core processors have become more popular in consumer hardware, scientists from a range of fields are making use of distributed and grid-based architectures to support their data processing requirements. As data volumes and computational requirements become increasingly large, the software applications facilitating the storage, analysis and interpretation of data become central to the scientific process.

Geographic Information Systems (GIS) are one example of an area in which vast quantities of data and data processing play a key role. GIS make use of a wide variety of geographically referenced data in order to support the modelling, analysis and understanding of the world around us. Despite the complexity of GIS, and the focal role of data in their design, GIS do not empower data, but instead provide a fixed data processing pipeline whereby scientists analyse and visualise the results of data manipulation in order to gain further data or information.

In contrast, Dynamic Data Driven Application Systems (DDDAS) use data obtained from measurements to inform an internal model, which in turn dictates how further measurements may be taken. This continuous loop of measurement and modelling empowers data in order to gradually evolve the measurement and modelling process. DDDAS are thus *data driven*, in that data drives their runtime behaviour, although the extent of data's influence on DDDAS behaviour is currently limited to controlling their execution flow and parameterisation.

The work described in this thesis does not aim to design and implement GIS or DDDAS solutions. However, the focal role of data in such systems makes GIS and DDDAS interesting candidates for novel data driven technologies. Furthermore, a common need for visualising models and analyses of real world problems make the concepts modelled by GIS and DDDAS more accessible as examples. This work has been motivated by a range of examples from both GIS and DDDAS, which have also been used to illustrate the capabilities and potential applications of the technologies introduced during the course of this work. The thesis will consequently contain examples from both GIS and DDDAS contexts.

Modern computer games are data, computation and visualisation intensive software applications. Growing consumer demands, consumer hardware capabilities, and market share competition, place increasing pressure on game development studios to produce more immersive, interactive and visually detailed game titles. At the same time, decreasing budgets and development cycles lead to a need for more productive software and content development methodologies. In the past, game developers have balanced data processing, runtime behaviour, detailed rendering and player interaction through the use of low level programming and hand written assembly optimisations. More recently, a wide range of hardware solutions have allowed developers to adopt more mainstream software development paradigms. Today's game studios release a series of similar titles based on the same underlying game engine, but with their individual story lines, content and player experiences tailored to each release.

While game engines maintain their focus on optimised runtime functionality, today's development of such solutions relies on a range of software methodologies including object oriented implementation and integration languages, programming patterns and third party libraries. Modern game engines are also making widespread use of data driven technologies in order to empower title specific data. In certain engines, game data is even used to drive the structure, composition and configuration of software using a range of declarative data notations. Such *data driven programming* allows the functionality of game titles to vary dramatically according to data, without the need to modify or even recompile the underlying game engine.

By empowering data to include control over software capabilities, functionality and runtime behaviour, the data driven programming techniques supported by modern game engines represent an amalgamation of recent developments in both the evolving role of data in applications, and increasing abstractions in today's software development methodologies.

This thesis introduces the Fluid project, describing the evolution and eventual design and implementation of a novel, data driven component based software framework. The Fluid framework represents the convergence of current trends in empowered data formats and increasingly abstract software development methodologies, by supporting the development of applications using small scale software components exhibiting a range of configurable data driven behaviour. The Fluid framework focuses on the flexibility of Fluid applications, providing an expressive XML based composition language supporting a number of high level manipulations from the object oriented programming paradigm. Although the Fluid framework has a wide range of potential applications, this thesis illustrates its potential with motivating examples from the GIS and DDDAS domains.

1.1 Contributions

This thesis contributes to the following fields:

Component Based Software Development The Fluid framework is a novel component based technology building upon contemporary component based approaches and game development technologies. The Fluid framework contributes to the field of Component Based Software Development (CBSD) by emphasising the flexibility, extensibility and expressiveness of application descriptions. Fluid's unique approach is facilitated by the use of high level abstractions and manipulations driving the assembly of small scale, highly configurable black box components.

Dynamic Data Driven Application Systems The work described in this thesis does not aim to develop DDDAS solutions. DDDAS are discussed as a related field due to their ability to incorporate data driven behaviour into their simulation and measurement processes. This thesis contributes to the field of DDDAS by identifying current limitations in the way DDDAS use data to drive runtime behaviour, and speculating upon how the application of the Fluid framework could enhance the capabilities of future DDDAS.

Visualisation of Spatiotemporal Scenes The thesis contributes to the visualisation of spatiotemporal scenes by introducing the PirateHat rendering library [3][1]. The PirateHat rendering library builds upon an implementation developed as part of this research in order to provide an open, platform independent alternative to an anticipated Microsoft technology.

1.2 Structure of the Thesis

This thesis is structured as follows:

Chapter 1 provides an introduction to the thesis, including a brief overview of the main themes covered by the research. Chapter 1 also includes an outline of the structure of the thesis, and a brief outline of the material found in each chapter.

Chapter 2 describes the context of the work, presenting a number of concepts that form the basis of the material given in the later chapters. Chapter 2 first discusses the evolving role of data in applications, focussing on the dichotomy of data centric and data driven applications. Chapter 2 then provides an overview of popular software methodologies at various levels of abstraction, including technologies related to software implementation, integration, component based approaches and model driven software development. Finally, Chapter 2 gives

[1]The PirateHat rendering library was developed in collaboration with an undergraduate student, and formed the focus of his final year project.

an introduction to computer games technology, concentrating on the recent growth of data driven techniques for defining scene content, as well as its visualisation.

Chapter 3 provides a detailed description of the Fluid component framework, and its novel combination of game technologies with mainstream component based methodologies. Chapter 3 describes the motivation for introducing game technologies to component based software development. Chapter 3 then introduces the Fluid component framework, describing its fundamental technologies, component model, composition language and executable behaviour. An example geovisualisation application is presented before giving a summary of the framework's current design and implementation.

Chapter 4 discusses the research described in Chapter 3 within the context of the concepts introduced in Chapter 2. Chapter 4 discusses how this work relates to the evolving role of data in applications, including speculation on its uses in DDDAS. Chapter 4 also relates the technologies developed during the course of the Fluid project to current software methodologies, including low level implementation and integration languages, component based technologies, architectural technologies and software development approaches. Finally, Chapter 4 speculates upon the impact of this work on the games development field that initially inspired its design.

Chapter 5 concludes the thesis by providing an evaluation of the work introduced in Chapter 3, clearly stating the contributions of this work to the fields described in Chapter 2, and discussing potential opportunities for further work.

Chapter 2

Context

The distinction between software behaviour and data is becoming increasingly blurred. While data is commonly accepted as encompassing information, measurements, or the offline representation of runtime state, data is becoming progressively empowered to incorporate elements corresponding to executable behaviour. The proliferation of object oriented methodologies has led to a clear separation of state and behaviour in software design and implementation. At the opposite extreme, interpreted languages relax the distinction between software and data: the definition of runtime behaviour may be represented using high level notations that form part of an application's runtime input and output; similarly, application data may be represented using a number of assignment statements that, when executed, perform the task of data input. Between these two extremes of separation and integration lies a scale of varying capabilities for modern data representations. Dynamic data driven application systems, component based methodologies, and model driven approaches provide a range of abstractions, at different scales of granularity, for allowing data to drive the runtime behaviour of software.

This chapter provides an overview for a number of relevant concepts, which collectively form the context and motivation for the material presented in the later chapters. The work presented in these later chapters describe a multidisciplinary project spanning a number of domains as illustrated in Figure 2.1. Sections 2.1, 2.2, and 2.3 do not embody a complete review of each concept presented in Figure 2.1. This chapter will instead define the research, methodologies and technologies deemed most relevant to the work, while including references for the interested reader.

Section 2.1 provides an overview of the evolving role of data in applications. Many applications from a wide range of fields include a substantial data processing element; Geographic Information Systems (GIS) are used as a motivational example. In the majority of cases, such data processing consists of a number of fixed operations deriving from a static software design. However, modern

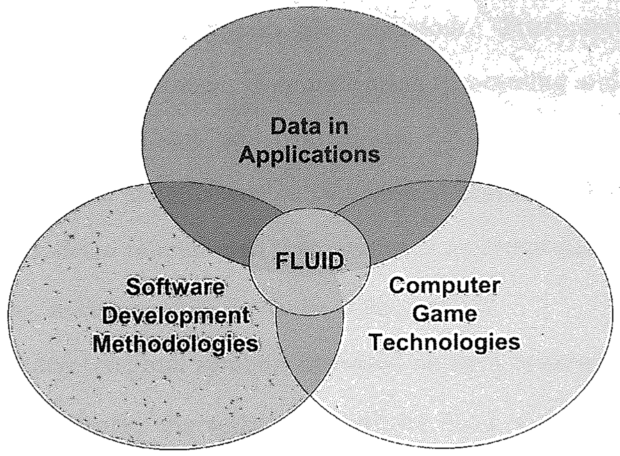

Data in Applications
FLUID
Software Development Methodologies
Computer Game Technologies

Figure 2.1: A Venn diagram including the domains contributing to the project.

data representations present a range of opportunities to derive additional information during data processing operations. This approach is embodied by the Dynamic Data Driven Application Systems (DDDAS) paradigm. Subsequently, elements from the GIS and DDDAS fields are used as examples in order to illustrate a variety of concepts, ideas, and techniques.

Section 2.2 describes a number of software development methodologies and technologies, with a particular focus on techniques that facilitate software implementation, composition and reuse in a modular fashion. This section includes a description of the object oriented (OO) paradigm, familiar to most software programmers today. The OO paradigm also shares a number of concepts with object based data models in GIS. Techniques are ordered according to their level of abstraction, moving from low level software implementation to the integration of existing implementations, through the selection of compositions to forming high level software behaviour specifications at the architectural level.

Finally, Section 2.3 discusses the role of computer games technology within the context of software engineering methodologies and the use of data in today's applications. This section introduces a number of hardware and software technologies that are being used to empower data in order to develop applications that are flexible to changing requirements, and capable of producing highly detailed, immersive visualisations at interactive frame rates. The discussion includes a brief overview of how these methods have evolved, and how they are applied today.

Section 2.4 gives a summary discussion of the material presented in Sections 2.1, 2.2, and 2.3.

2.1 Data in Applications

Data storage and processing play a central role in a wide range of applications; indeed, almost all software solutions incorporate some element of data use, from simple data storage representations to powerful notations capable of driving application behaviour. Furthermore, a diverse range of technologies related to data formats, storage, communication, processing and manipulation continue to emerge and evolve due to academic and industrial efforts.

Many definitions of data include an emphasis on facts and information. For example, the following dictionary definition also includes a reference to the computer representation of data [4]:

1 a series of observations, measurements, or facts; information

2 the numbers, digits, characters, and symbols operated on by a computer

Within the context of this work, the term *data* will be used when referring to the offline or runtime representation of measurements, facts or information from a range of attribute domains. Runtime data is synonymous with software state, and will be represented during software execution by program variables corresponding to multiple bytes of computer memory. Runtime data may be written to operating system files, stored in databases or transmitted to other offline media, at which point they become offline data. While many definitions of what data *is* are very similar, the role of data in applications is continually evolving, as illustrated by Figure 2.2.

Early applications incorporated their data as part of the software code. Application data was represented by concrete values forming part of the implementation, and could not be modified without requiring the software to be recompiled. The application's runtime behaviour and data were inseparable: software execution was partly defined by its data, and the data could not be separately accessed or modified.

Later applications separated their runtime behaviour from the data to be processed. Application functionality was implemented using programming languages, while data was held in separate files. However, the applications and their data were still tightly coupled due to the use of proprietary and opaque data formats, and the resulting requirement for dedicated procedures to read and write the data files. Software implementation methodologies have since evolved to further empower application data. For example, the object oriented methodology places a particular emphasis on the separation of runtime state and behaviour. Runtime state is *encapsulated* by object interfaces, which define appropriate access methods and manipulations, thus enforcing that the data remains valid and meaningful within the context of its corresponding application domain.

Interpreted languages blur the lines between behaviour and state by allowing data to contain executable code, and for such code to be generated procedurally at runtime. However, it is important to distinguish application data from those files containing interpreted language code, which will be referred to as *scripts* (interpreted languages are often referred to as *scripting languages*). The primary purpose of application data is to contain a collection of values, which together constitute the input, output or current state of the application's various runtime processes. By contrast, executable scripts

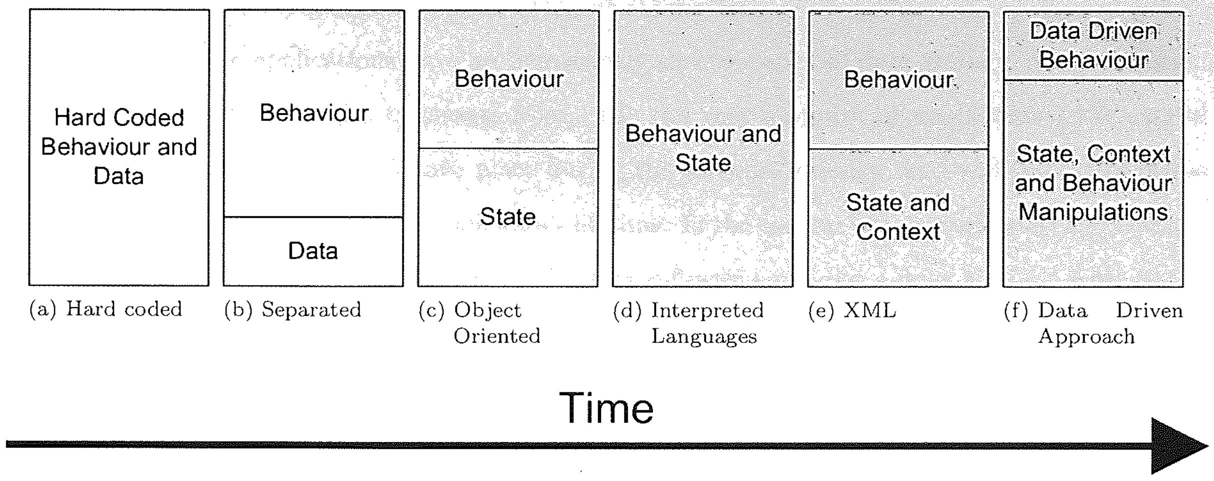


Figure 2.2: The evolving role of data in applications over time.

describe runtime behaviour within the context of a given scripting language and are intended for execution.

While implementation methodologies have been providing improved support for data representation and manipulation, data formats have been incorporating increasing degrees of contextual and behavioural information. eXtensible Markup Language (XML) and its related technologies have recently emerged as a family of extensible data formats that are becoming ever more popular in a wide variety of application domains. Whereas early data formats had to conform to closed standards focussing on particular applications, XML is an open format that may be extended to suit a range of uses.

Building upon the fundamental concepts of XML, XML *schema* may be used to describe a given data model, incorporating how data is to be represented as data types, and what data constructs are valid or meaningful within a particular context. Subsequently, XML schema are widely used to validate XML data documents against their corresponding data models. In addition, many XML schema are also comprehensive enough to derive (de)serialisation behaviour from the described data model, facilitating the automated generation of corresponding software implementations using a variety of languages via *data binding* [5, 6, 7]. Finally, the eXtensible Stylesheet Language Transformations (XSLT) format, which is itself derived from XML, allows a variety of transformations from one data representation to another to be described.

The XML family of data technologies is perhaps the most popular example of empowered data formats in use today. Many previous data representations have relied upon tightly coupled software applications to provide contextual information, various rules regarding data representation, and manipulation behaviours. By contrast, XML allows data to exist independent of a particular software implementation by describing the context, data model, and meaningful behaviours alongside the data itself.

Modern software applications have an increasingly diverse range of data storage, communication and manipulation technologies to choose from. For the vast majority of applications, the careful selection of such technologies will take place during the software's design, and will thereafter define the role of data for the duration of the software's lifetime. In the context of this work, such applications are referred to as being *data centric*: the application software embodies a data *pipeline*, with one or more data inputs, a variety of data outputs, and a fixed data process in between. Data centric applications are discussed in greater detail in Section 2.1.1.

By contrast, a number of recent technologies allow data to incorporate elements which explicitly determine, drive or otherwise significantly influence an application's runtime behaviour. Such *data driven* software responds to empowered data formats, such as XML, by modifying runtime capabilities in order to meet initial or even dynamically changing functional and non functional requirements as described by the data. Data driven software embodies a data *engine*, with a data input that describes the software's runtime parameterisation, structure, or architecture, and thus the capabilities of the software as a whole. Section 2.1.2 provides further description of data driven applications.

2.1.1 Data Centric Applications

In data centric applications, data forms the input and output of a data processing pipeline, as illustrated by Figure 2.3. Data centric applications provide fixed functionality: the runtime capabilities of the software will not change over time unless its underlying implementation is appropriately modified and recompiled.

In many data centric applications, the data itself is immutable, and the application focusses on data processing and manipulation in order to produce a range of data or informational outputs. Alternatively, data centric applications may provide a transformational pipeline whereby data comprises the current state of a particular application domain. In such cases, the software will be responsible for maintaining the current state via various data modifications.

The GIS field provides a range of examples of data centric applications with both static (immutable) and dynamic (mutable) data roles. Data forms a central part of all GIS applications, which collectively

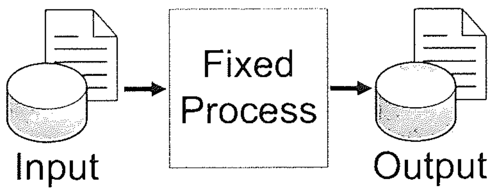


Figure 2.3: A data centric application, with data forming the input and output of a fixed functionality process.

focus on the efficient storage and informative modelling and manipulation of geographically referenced data. The importance of data in GIS is highlighted by the following definition [8]:

> A geographic information system is a computer-based information system that enables capture, modelling, storage, retrieval, sharing, manipulation, analysis, and presentation of geographically referenced data.

All GIS make use of one or more databases, which are responsible for the efficient storage of a variety of geographically referenced data. This data forms a fundamental part of all operations available to the GIS user. Each GIS will also define a *data model*, which stipulates how the real world is to be represented within the GIS. The data model employed by a given GIS will define how data is to be stored in the database, how data is read from and written to the database, what analysis and manipulations are meaningful, and how they are performed via the GIS software's user interface.

Although traditional GIS provide a fixed range of data processing behaviours over immutable data sources, there is a clear movement toward more dynamic data representations and functionality. This trend can already be seen in the use of visualisation technologies in GIS. Current efforts in *geovisualisation* focus on the use of dynamic behaviour, interactivity and multimedia [9]. Geovisualisation is a branch of scientific visualisation using computer systems to gain insight into and understanding of geospatial information [9]. Geovisualisation techniques extend the map metaphor in order to produce human-computer interfaces that are more intuitive and more expressive. All GIS include support for the display of static two-dimensional representations, which can be used to convey a variety of information as static maps. However, visualisations including three dimensional and virtual reality elements, animation, non visual output, and interactive feedback are becoming more common [10, 11, 12].

Geovisualisation may be applied for a variety of reasons, and may be employed as part of processes involving field or object based operations. In such cases, geovisualisation may be an important step in the initial input and validation of data, as well as during its subsequent manipulation as part of an ongoing study. Geovisualisation may be used to communicate the results of such manipulations in order to inform [13] or assess the impact of proposed developments or changes [14]. However, it is important to consider the effect of such technologies on participating lay persons, professionals and members of the public; several factors, particularly the added realism of geovisualisations over traditional cartographic displays, have been shown to add considerable bias to the planning process [15, 16].

Data plays a key role in GIS, where an extensive range of functionality regarding the representation, storage, transmission, manipulation and visualisation of geographically referenced data is determined by an appropriate data model. Despite data processing forming a substantial part of GIS applications, GIS software remains *data centric*, in that the behaviour facilitating data manipulation is fixed when the software is compiled, and cannot be changed without recompilation. Unlike *data driven* applications, where the runtime behaviour of software is driven by the contents of application data, the roles of data and behaviour in GIS are established by software implementation, with neither being able to adapt or evolve over time. Current trends in geovisualisation provide an example of where the field of GIS is beginning to move toward a more object based, dynamic and integrated model of the world around us. However, while research efforts in GIS incorporate an increasing range of dynamic behaviours, a substantially data driven GIS has yet to be realised.

2.1.2 Data Driven Applications

Data driven is a popular term in academia with ambiguous meaning, often referring to data playing a central role in an application's functionality or to an emphasis on data processing. As described above, in the context of this work, such applications are not considered to be data driven, and will instead be referred to as *data centric*.

By contrast, an application is *data driven* when its data input significantly influences or *determines runtime behaviour*. *Data driven programming* is a programming paradigm used to enable data driven approaches; it represents a movement towards higher-level, more declarative notations for describing application behaviour. Data driven programming advocates moving design complexity away from fixed software implementations and into data representations that are convenient for humans to maintain and manipulate with appropriate tools. *Data driven applications* are responsible for translating those representations into runtime behaviour, so that application users or client applications may influence, steer or even determine application behaviour via data manipulations. These basic concepts of the data driven approach are emphasised by the following definition [17]:

> When doing data-driven programming, one clearly distinguishes code from the data structures on which it acts, and designs both so that one can make changes to the logic of the program by editing not the code but the data structure.

The data driven approach to software development has been adopted by the relatively new DDDAS paradigm. Core to the DDDAS paradigm is the emphasis on steering application behaviour using data driven methods, as illustrated by the following definition [18]:

> Dynamic Data Driven Applications Systems entails the ability to incorporate additional data into an executing application, and in reverse, the ability of applications to dynamically steer the measurement process.

DDDAS aim to improve the accuracy of the analysis, prediction and modelling of dynamic systems. While traditional simulations rely on functionality and inputs that are fixed at application execution, the approach embodied by the DDDAS paradigm is to refine or enhance the simulation by injecting addition data into the model at runtime. DDDAS are thus able to form more complete models as additional data becomes available, leading to more accurate analysis and predictions. In addition, a more informed model may be able to enhance the measurement processes by adjusting their parameters in order to optimise the usefulness of their output. Such capabilities are highly desirable where measurements are difficult to perform, time consuming, time intensive, or expensive.

The DDDAS paradigm creates a number of new requirements for data content and processing. DDDAS requires applications to accept data at runtime, and to adapt their behaviour in order to be dynamically steered by such data. It is therefore important that application data contains relevant information that can be used to drive application adaptations. Similarly, applications must include additional routines that are able to locate and interpret those data elements that indicate a requirement for such change. In order to modify their runtime functionality and behaviour dynamically, applications must also incorporate a representation of their current capabilities, and have access to a range of operations that are able to modify such capabilities. In order to meet these requirements, and many others not mentioned here, DDDAS is the focus of a range of multidisciplinary research projects funded by the National Science Foundation (NSF), and the subject of numerous conference workshops[1].

A wide variety of DDDAS applications exist. In order to provide an illustration of the scope and focus of current NSF funded projects, the following subset of examples is adapted from an online list of current areas of interest[2]:

Fire propagation prediction and management Research in this area can be applied to modelling fires in both open and enclosed spaces, such as wild and forest fires, in order to aid the evacuation effort, fire containment, and the distribution of fire response personnel. The application of such technologies is demonstrated by existing DDDAS modelling urban [19] and wild fires [20]. The former applies an agent based approach to analyse existing fire data in order to consider the effect of improved building planning, while the latter uses numerical methods in order to integrate data from remote sensors to produce 2D and 3D visualisations.

Advanced Driving Assistance System The driving assistance system continuously predicts vehicle motion by model-based real time simulation incorporating measurement data such as the steering angle, vehicle accelerations and yaw rate. Vision may be another useful input. On a larger scale, DDDAS are also considering the monitoring and management of traffic systems, with a particular focus on emergency response scenarios [21].

Biological 'real time' experiments can involve time scales ranging from nanoseconds for molecular and sub-molecular dynamic processes to years and decades for ecological changes. For

[1]`http://www.nsf.gov/cise/cns/dddas/`

[2]`http://www.nsf.gov/cise/cns//dddas/DDDAS_Appendix.jsp`

example, in experiments recording neural activity, where the dynamics occur at different speeds, it could be very useful in general to have 'smart devices' that would shape an experiment in real time.

Hydro-complexity - Weather, Water and Pollution Weather processes are modelled at large spatial scales; hydrologic processes are modelled at smaller spatial scales, and groundwater pollution is modelled at even smaller spatial scales and much longer time scales. One of the greatest challenges in "water process" simulation is to find a practical way to connect modules that operate and require inputs of dynamic data at vastly different scales. DDDAS offers vast opportunities for addressing non-linearities in this highly interactive system. Research exploring the use of roaming sensors in order to locate and identify water contaminants [22] provides an example application.

In all cases, measurement and simulation processes form a synergistic feedback loop that dynamically adjusts the runtime behaviour of the DDDAS software, as illustrated in Figure 2.4b. By contrast, current GIS are typically limited to fixed functionality as shown in Figure 2.4a.

2.1.3 Summary of Data in Applications

The role of data in applications continues to evolve with the introduction of new software development methodologies, technologies, and application domains. As software applications incorporate increasing amounts of dynamic behaviour and information, there is a necessity for their implementations to become more adaptable to dynamically changing functional requirements.

Data centric applications are those whose runtime capabilities are fixed during the design phase of software development. While data centric applications incorporate extensive data processing facilities, their concrete implementations prevent the data formats or processing routines from evolving in

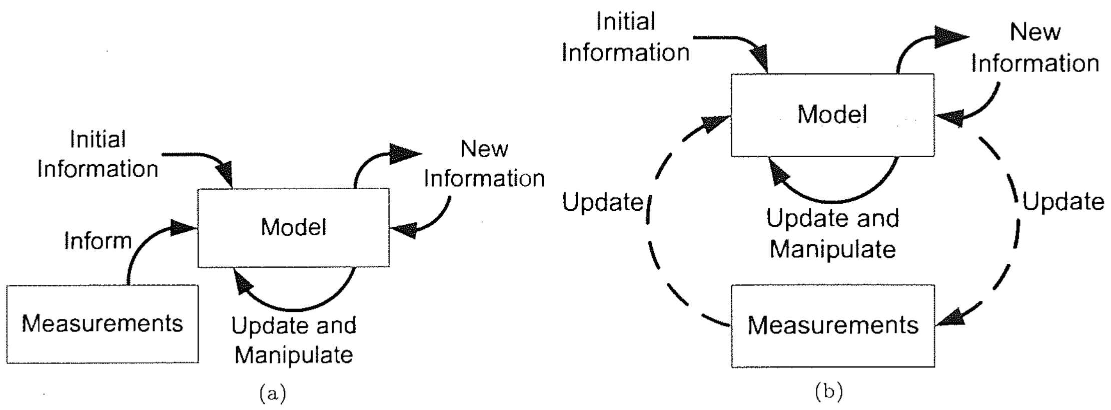


Figure 2.4: An illustration of the functionality available to GIS (a) and DDDAS (b). Dashed lines in (b) denote the synergistic feedback loop formed by DDDAS measurement and simulation.

response to changing storage or additional operational requirements. Traditional GIS provide an example of data centric applications.

Conversely, data driven approaches allow software applications to dynamically modify their runtime capabilities in response to changing requirements. Certain data technologies, including the XML family of extensible data formats, support the development of data driven software by incorporating additional information that may be used to drive, determine or significantly influence runtime application behaviour. DDDAS are already making widespread use of data driven techniques. However the application of data driven technologies in DDDAS is currently limited to a classical interpretation of *data driven*, in that the software's runtime behaviour is changed via parameterisation, and not through adaptation of the software itself. DDDAS adapt their runtime behaviour by dynamically modifying the flow of software execution and altering the parameterisation of their internal models, simulations and operations. While these modifications allow DDDAS to change their behaviour in response to runtime data, such changes are anticipated during software design, and are consequently supported by their implementations via the exposure of dedicated branch and loop conditions and appropriate parameterisations.

Although classical approaches to data driven software development facilitate dynamically changing software behaviour, they are incapable of supporting truly adaptive applications that respond to changing requirements by incorporating new functionality. A DDDAS supporting more extensive dynamic adaptation would be able to optimise its capabilities at runtime according to *unforeseen* changes in its data processing requirements, potentially leading to more efficient and cost effective applications. However, in order to achieve such dynamic flexibility, the predetermined branch and loop conditions of existing DDDAS must be replaced with more flexible, extensible and adaptive software development methodologies. In particular, a deployment concept providing variable implementation granularity would allow for the progressive, flexible definition of overall application functionality. Meanwhile, higher level notations should be available to drive the (re)configuration of application functionality.

Sections 2.2 and 2.3 describe a number of technologies and approaches that may be able to introduce a novel degree of dynamically flexible and reconfigurable behaviour to future adaptive software applications.

2.2 Software Development Methodologies

Software engineering is the process of developing software solutions, typically encompassing a number of phases such as requirements elicitation, software design, software implementation and solution deployment. The number, ordering and naming of these phases will depend on the software development methodology in use - for example, the Rational Unified Process [23] (RUP) includes four phases denoted Inception, Elaboration, Construction and Transition. However, the overall process of software engineering remains the same: the client's problem is carefully documented, a solution to the problem is encapsulated by a software design, the software design is implemented as a software solution, and finally the software solution is delivered to the client.

A wide range of existing solutions, technologies, practices and methodologies exist to aid software engineers in the development of today's increasingly complex and large scale software solutions. Of particular interest are a selection of methodologies supporting software implementation and composition at various levels of abstraction, as described in the following sections and as illustrated by Table 2.1.

Programming Languages are responsible for providing expressive notations for the definition of low level operations by software programmers. Such languages are used to implement the fundamental software solutions, libraries and frameworks upon which higher level abstractions may rely.

Each programming language will stipulate its own notations, conventions and both syntactic and semantic standards, which must be adhered to in order to develop valid software solutions. Irrespective of the language used, valid implementations are invariably translated to machine code during a compilation process, and are subsequently executed at runtime by an appropriate hardware platform. A diverse range of paradigms exist, supporting a variety of abstractions including procedural, object oriented and functional programming concepts to name but a few. In the context of the work presented in this and following chapters, the object oriented paradigm is the most popular, most commonly used, and hence most relevant paradigm to consider in depth; see Section 2.2.1 for further details.

Scripting Languages in the context of those concepts presented in Table 2.1, are also known as *glue* or *integration* languages, and are commonly used to compose, connect, or *integrate* existing

Level	**Languages**	**Technologies**
Specification Level	Architecture Description Languages	Component Technologies
Selection Level	Composition Languages	
Integration Level	Scripting Languages	Software Libraries and Frameworks
Implementation Level	Programming Languages	

Table 2.1: Software engineering methodologies and technologies supporting implementation composition and abstraction. Note that the Unified Modelling Language (UML) provides abstractions at all four levels in the table, while model driven techniques often encompass all four levels as part of its top down approach.

modular solutions that have been implemented using lower level programming methodologies such as those described in Section 2.2.1. *Composition* here refers to the assembly and resulting structure of software implementations, while *connection* is an association relationship formed between two or more implementations, typically via function calls. Meanwhile, *integration* is the use of aggregation and association to bring otherwise independent technologies together within the runtime environment provided by the scripting language. In order to facilitate such compositions, scripting languages commonly incorporate a range of higher level abstractions that can be used to provide integrations, interfaces, or adaptations for a diverse range of distinct implementations.

While not all scripting languages are interpreted, the particular features of interpreted languages make them ideal candidates for scripting. Their expressive syntax, flexible runtime semantics, platform independence, and native support for numerous abstract data structures and algorithms, have led to the widespread use of interpreted languages to provide scripting functionality. Section 2.2.2 provides a descriptive introduction to interpreted languages, with an emphasis on their use as tools for scripting and implementation integration.

Scripting languages are employed to integrate existing implementations, with the integration process operating over concrete components and their corresponding interfaces; these are commonly software libraries and Application Programmer Interfaces (APIs) implemented using programming languages. Scripting languages are used to facilitate the communication between independent interfaces, in some cases including interfaces written in different programming languages. The role of scripting languages is therefore one of translation and communication: a message transmitted by one component will be translated by the scripting language to some intermediate representation, communicated to the receiving component's location, and then translated to an appropriate format to be understood by its API.

Composition Languages In contrast to scripting languages, composition languages stipulate which components are taking part in a composition, and how the components are composed. They can thus be considered declarative notations, which are commonly supported by the facilities provided by imperative scripting languages. Composition languages provide a dedicated language interface to a particular set of composition mechanisms. These mechanisms are defined by a *component model*, which stipulates what components, compositions, connections, and communications are valid for a given application domain. In the context of composition languages, *composition* has a more complex meaning than the compositions supported by scripting languages as discussed above. The exact semantics of component composition depend upon the component model in use: composition may relate to the structure of interconnected components, but may also introduce additional concepts such as component *composites*. Furthermore, the result of component composition may be a new (possibly dynamically defined) component. Meanwhile, component *communication* is the passing of both data and control flow between components, while component *connection* is the act of binding the services provided by one component to the services required by another so that communication may take place. The component model is implemented by a *component framework*, which facilitates component deployment, composition, connection and communication at runtime. Composition languages are expressive notations that *drive* component framework operations; they typically rely on programming and scripting languages to provide lower level application functionality and integration respectively. Composition languages, as well as their related technologies, are described in greater detail in Section 2.2.3.

Architecture Description Languages represent the highest level of abstraction considered in the following sections. Architecture description languages (ADLs) build upon the concepts available to composition languages in order to describe large-scale component based applications while explicitly focussing on their architectural features. In particular, ADLs commonly provide two main abstractions for describing component based software: components, as described in Section 2.2.3, and connectors. Section 2.2.4 provides an overview of ADLs in the context of this work.

The specification, selection, and integration technologies presented in Table 2.1 build upon implementation technologies in order to provide successively higher level abstractions for the bottom-up description of software solutions. A similar approach is taken by *model driven* methodologies; whereas traditional software development involves the manual process of deriving low level implementations from software design, model driven techniques concentrate on the automatic transformation of abstract models to derive software code. While traditional software development methodologies endeavour to support the process of manually developing software solutions as low level implementations, model driven approaches ultimately aim to automate the task of code generation from abstract notations encompassing software design. A brief overview of the model driven software development methodology is presented in Section 2.2.5.

2.2.1 Object Oriented Methodology

The object oriented programming methodology defines software behaviour as the interaction of a number of *objects*, each of which corresponds to an entity in the problem domain. Every object is responsible for maintaining a valid representation of its corresponding entity, including state and behaviour. In this context, a *valid* representation is one where the object's state and behaviour are reasonable representations of what one would expect the state and behaviour of the object's counterpart in the problem domain to be.

A class is the unit of definition used to describe object oriented software functionality. Each class exemplifies a single particular type of entity in the problem domain. For example, the set of all vehicles would be represented by a corresponding collection of vehicle classes, with each type of vehicle (cars, buses, motorbikes and so on) being represented by a single distinct subclass. Class descriptions are *static* specifications designed using object oriented methodologies, tools and notations, with the Unified Modelling Language being one of the most widespread examples. Class designs are implemented using object oriented programming languages such as C++ or Java.

By contrast, an object is the dynamic (runtime) manifestation of a class, with each object representing a single instance of the concept defined by its corresponding class description. Objects only exist during software execution, where each object has a limited lifetime starting with the object's instantiation and terminating when the object is destroyed. During its lifetime, each object is allocated within a portion of application memory, which will also include a representation of the object's constituent runtime state. Object state may change as a result of object interactions, and it is the the culmination of these interactions and subsequent state changes which together result in overall application runtime behaviour.

Object oriented design supports two main relationships between classes: aggregation and inheritance. Aggregation embodies a 'has-a' or 'whole-part' relationship between owning and constituent classes, allowing a given class instance to contain one or more instances of other classes. Aggregation allows complex concepts to be described as the combination of more simple concepts. For example, a `Car` class description may use aggregation to describe the `Car` as the assembly of an `Engine` instance, a `Chassis` instance, four `Wheel` instances, and so on. The use of aggregation also leads to a hierarchy of encapsulated state and behaviour, where each enclosing class is responsible for its contained parts.

Classes may also be related via inheritance, which embodies a 'is-a' relationship between parent and child classes, where a child class will automatically include or inherit the members of its parent class. For example, a `Car` is-a type of `Vehicle`, and so the `Car` class will inherit all of the attributes and methods associated with the `Vehicle` class description. Inheritance therefore provides a mechanism that allows a subclass to be described as an extension or specialization of its parent class. Extending subclasses may describe attributes and methods in addition to those inherited from its parent, while specialising subclasses may provide their own descriptions for members of the parent class, thus *overriding* their definitions. Subclasses may both specialise and extend their parent types at the same time, by overriding some members and extending others.

This combination of class inheritance and overriding facilitate *polymorphism*, where the runtime behaviour of a given method invocation varies according to which class the method definition belongs to. For example, invoking a `boardVehicle` method may involve opening and closing doors for `Car` classes, paying a driver for `Bus` classes, lifting a stand for `Motorbike` classes, and so on. Polymorphic behaviour typically occurs when a function or procedure expecting a parent class parameter is invoked using an instance of a derived (child) class. This form of *type substitutability* is supported because of the inheritance relationship between the parent and child class descriptions: a `Car` is-a `Vehicle`, and will inherit the functionality of its parent type, so any action performed on an expected `Vehicle` instance should also be valid if performed on an instance of the `Car` class. However, if the `Car` class description *overrides* the definition of one or more `Vehicle` class methods, then performing any action on a `Car` instance may result in differing (polymorphic) behaviour than that expected from a `Vehicle` instance.

The OO methodology thus provides an abstraction of software functionality at the granularity of archetypes (classes) and instances (objects) that closely resemble concepts from a particular problem domain. Classes and objects encapsulate software state and behaviour, thus hiding certain implementation details, while select operations may be exposed via class and object interfaces. However, certain circumstances require a coarser level of granularity than that offered by classes and objects. For example, it may be desirable to encapsulate a number of classes belonging to a software implementation, while exposing others, in order to hide certain implementation details and produce a more simple and coherent interface to the implementation's collective functionality. In such cases, *software libraries* may be used to provide coarse grained encapsulations for implementation development and distribution.

Software libraries contain the compiled form of software implementations, and may encapsulate a range of functionality from single classes to entire applications, although most libraries perform a single coherent role via a well-defined API. Software libraries thus represent a movement toward more coarse grain implementation hiding and reuse than that offered by class abstractions. A wide variety of applications may be developed by utilizing the functionality provided by, and thus building upon, library implementations. There are two main types of software library, which are distinguished by how their functionality is combined, or *linked*, with other software implementations. Static libraries are linked with other software implementations by a *linker* program as part of the compilation process, while dynamic libraries are typically linked during application execution. Both static and dynamic software libraries promote implementation reuse on various scales, from the granularity of individual software components, to applications and software frameworks.

Software frameworks provide semi-complete, reusable implementations of software applications. A software framework typically consists of a large number of classes, some of which may be abstract, which realise a variety of software patterns. Together, these patterns collectively form a partial implementation that may be completed by third parties in order to form concrete applications. Frameworks facilitate third party development by providing clear and well defined interfaces for customisation or extension, as illustrated in Figure 2.5.

Framework based software development focusses on massive design and implementation reuse by providing an existing backbone of domain specific design, implementation and documentation. Frameworks thus embody a wealth of domain knowledge, and may include any number of system-wide design decisions and optimisations. This encapsulation of domain knowledge and core run time functionality allows software frameworks to deliver a stable and coherent backbone upon which further development may be based.

Third parties create software applications by extending or customizing frameworks at predefined locations. These customisation points typically consist of a number of abstract interfaces for which derived concrete implementations may be defined, thus introducing custom polymorphic functionality into the framework's run time behaviour. Custom implementations are typically provided in the form of dynamically linked libraries, which are often known as *plugins* in the context of software frameworks. Some software frameworks provide default implementations for their customisation points, potentially reducing the number of changes required for lightweight extensions, while others require a number of customisations before they can be used at all. Customisation points reduce complexity by providing clear, restricted interfaces that remove design decisions from application developers. They are thus responsible for limiting the amount of understanding required in order to develop large-scale software applications, and to manipulate their overall structure and run time behaviour.

For example, the Eclipse[3] Integrated Development Environment (IDE) provides an extensible framework that may be extended by third parties. The Eclipse platform supports a number of *Eclipse plugins*, which work together to form a seamless programming environment, performing a variety of tasks such as dynamic syntax checking and highlighting, and software compilation; Eclipse plugins

[3]www.eclipse.org

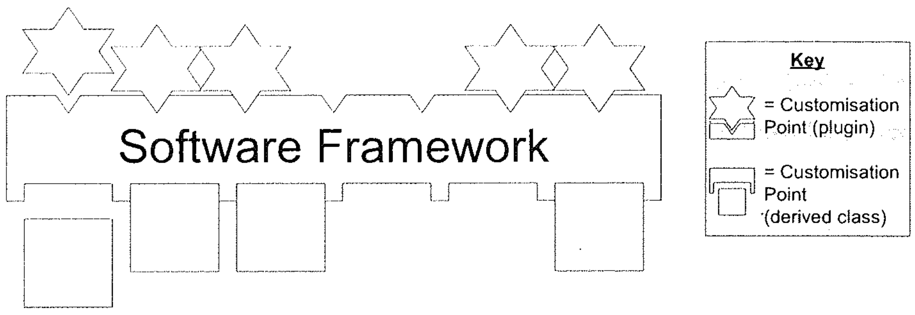

Software Framework
Key
= Customisation Point (plugin)
= Customisation Point (derived class)

Figure 2.5: An illustration of the customisation points provided by software frameworks.

may also extend the Eclipse GUI to include additional controls. Developers are able to extend Eclipse by wrapping their custom functionality as one or more Eclipse plugins, which must conform to an exposed plugin contract. The Eclipse plugin contract provides a powerful customisation point for the Eclipse platform, allowing developers to both extend and adapt Eclipse to meet their own development requirements.

Despite the popularity of object oriented approaches, OOD and OOP have often been criticised for failing to deliver the reusability, flexibility and extensibility for which they were first introduced. Some argue that a focus on white box reuse makes it difficult to extend OO software without first fully understanding existing code [24]. It is also argued that OO design does not exhibit a clear separation of computational and compositional concerns, leading to implementations that cannot cope with the rapidly changing requirements of present day applications [25]. Further criticism highlights a number of features of OO approaches that may hinder the modification of existing implementations [26]: OO analysis and design are largely domain driven, consequently typically domain specific, and reuse of existing implementations occurs much too late in the development process; interfaces and interaction protocols are overly verbose; and software architecture is implicitly described by the connections between its constituent classes or runtime objects.

Implementation technologies typically support the development of software solutions at a low level of abstraction. The flexibility of such solutions is often limited: the purpose of implementation technologies is to map high level abstractions embodied by software design to corresponding lower level definitions targeting a particular hardware platform. The resulting software code is of a concrete nature due to its close correspondence to a concrete and inflexible execution environment. As discussed in Section 2.1.2, the existing use of low level implementation methodologies offers limited support for dynamic flexibility and adaptability of DDDAS functionality. By contrast, a much greater degree of dynamic flexibility is supported by interpreted languages, as described by the following section.

2.2.2 Interpreted Languages

Interpreted languages provide a powerful alternative for writing application software, by incorporating a number of high-level abstractions offering a wide range of benefits to application developers. Many interpreted languages allow a combination of interpreted and statically compiled languages to be used together, or *integrated*, as part of the same application.

When using statically compiled programming languages such as C, C++ or Ada, program code is processed once by a *compiler*, resulting in machine code that may be executed by a particular hardware platform. By contrast, interpreted languages such as Java, Python and Lua are compiled to an intermediate form known as *byte code*. Byte code consists of a sequence of rudimentary instructions, typically similar in both appearance and meaning to assembly code, which may be read and executed by an *interpreter* application, resulting in the intended run time behaviour. To this end, interpreters commonly provide a simulation of an underlying hardware platform or *virtual machine*.

Figure 2.6 presents an outline of the compile and execution lifetime of typical statically compiled language, while Figure 2.7 presents that of interpreted languages. The illustrations also highlight one of the major benefits of using interpreted languages: both code and byte code forms of interpreted software applications are platform independent. All one requires to execute a given interpreted language application on any given platform is a virtual machine compiled for that platform. Meanwhile, statically compiled languages must target a particular operating system and hardware platform during the compilation process. Platform independence has contributed to the popularity of languages such as Java and JavaScript, which benefit from simple software-reuse facilitated by a standard language and libraries with no external platform dependencies. Contrast this with statically compiled languages and libraries, which often force application developers to acknowledge platform differences and dependencies.

The vast majority of interpreted languages may be used to write stand-alone applications, and are not necessarily restricted to small text-processing applications, but are increasingly being used to write medium to large scale software. For many applications, a number of interpreted languages rely upon the functionality provided by C and C++ libraries in order to benefit from their lower-level functionality and features. For such uses, some interpreted languages include interfaces for the purpose of communicating with other (statically compiled or interpreted) languages. For example, Lua modules may be developed using Lua code, or optionally deployed as C or C++ libraries. Lua includes facilities for accessing the functionality encapsulated by such modules in a uniform way.

Similarly, many software developers are recognising the benefits of having the high-level features of interpreted languages available as part of their applications. This has resulted in a number of applications *embedding* interpreted languages in order to offer their functionality to users and other developers. For example, some applications may include an interpreted command environment for

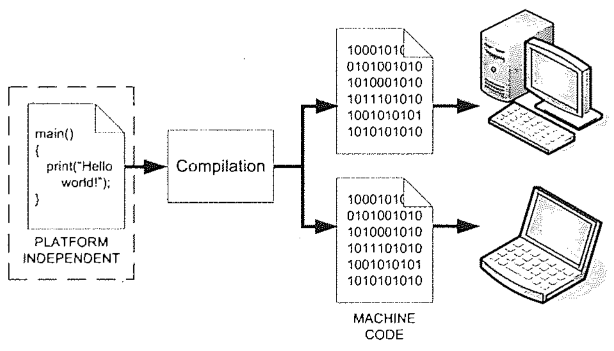


Figure 2.6: A illustration of the typical compile and execution lifetime of statically compiled languages.

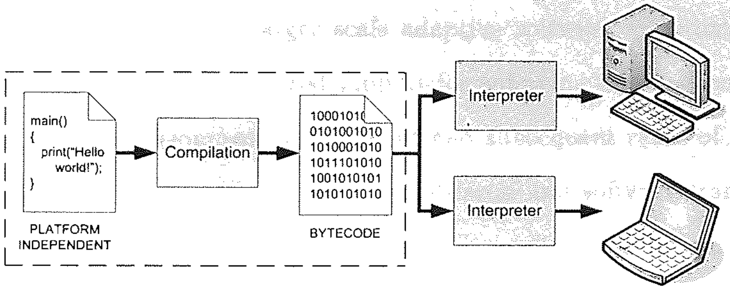


Figure 2.7: A illustration of the typical compile and execution lifetime of interpreted languages.

run time manipulation of program state, while others may use an interpreted language to support the use of initialisation scripts. These scripts will be executed by an application when it is first executed, and will typically assign initial values, invoke initialisation behaviour and so on. The interoperability of many interpreted languages has allowed software developers to bridge the gap between statically compiled and interpreted implementations, by supporting varying degrees of both languages and thus a mix of their individual features, functionality, benefits and drawbacks.

While interpreted languages offer powerful and abstract environments for the development of software solutions, they are not suitable for all applications. Dynamic byte code interpretation incurs an additional runtime cost, while many abstractions such as dynamic typing hide operations that the software developer may wish to control or constrain in favour of further runtime efficiency or additional type safety. The abstractions offered by many interpreted languages may come at the cost of control or runtime performance.

Interpreted languages allow software developers to make use of a wide range of high-level abstractions and implementations, improving the interoperability and re-use opportunities of their products, as well as increasing their own productivity. Software development may be simplified by removing low-level concerns from the development environment, using either those attributes common to all interpreted languages such as the virtual machine concept, or via language-specific features such as dynamic typing. A particular benefit of using interpreted languages is that their strengths may be combined with those of other languages: a developer may write the majority of a software application using Python, while implementing certain modules in C++ where additional runtime efficiency is required. Interpreted languages thus allow developers to *integrate* their higher-level attributes with those of lower-level implementation languages that must be statically compiled.

While implementation languages provide a mapping from high level software design to static and concrete software solutions, interpreted languages emphasise runtime flexibility and adaptability by supporting an execution environment with a higher level of abstraction than the target hardware platform. However, many interpreted languages also introduce a number of disadvantages that could prove prohibitive to the development of larger scale adaptive software applications. For example, interpreted languages often provide very limited support for enforcing information hiding and implementation encapsulation: the development, deployment and subsequent reuse of implementations is typically based on plaintext scripts, rather than closed libraries and software frameworks. The concepts described by such scripts are also often open to further modification by other executing scripts in the same program. Furthermore, many language features such as dynamic typing focus on software flexibility, and do not support the degree of static control or safety as provided by statically typed languages. While it is clear that the development of more adaptable software applications will require the use of more flexible technologies than those currently in use, the flexibility offered by interpreted languages may not be conducive to large scale software development or the careful reuse of existing implementations.

Between the two extremes of implementation technologies and interpreted languages, component based approaches offer an intermediate alternative allowing independent software implementations of varying granularity to be assembled using higher level notations. The resulting component based software offers both high level behaviour control and low level implementation flexibility, and thus has the potential to enhance the development of future adaptive software applications.

2.2.3 Component Based Software Development

Component Based Software Development (CBSD) is an established field of study within software engineering. CBSD is an active research area, attracting interest from many commercial and academic stakeholders, as illustrated by the diverse range of component based projects and products. The focus of the CBSD methodology is to facilitate the development of software applications via the selection, deployment and composition of prefabricated binary software components. The resulting software applications are commonly known as *component software applications*. CBSD builds upon a number related concepts from software engineering, such as software objects, software frameworks, and software modelling. At a basic level, CBSD may be considered similar to the object-oriented methodology, albeit at a higher level of abstraction and with coarser elements to be composed: the underlying idea is to define a collection of encapsulated software parts and to form new software applications by connecting them together via their interfaces.

The difference between OO programming and component based approaches can be subtle, and it is easy to confuse components with objects, and the composition of components with the assembly of objects in order to form complete applications. Indeed, in practice there may be little or no difference between some components and their corresponding OO implementations.

However, CBSD introduces a number of features, methods and technologies that collectively improve upon those available to the OO programming paradigm [27]. While OO implementations are commonly distributed as classes, libraries or frameworks, components typically consist of a collection of classes in order to deliver a single coherent concept. Furthermore, a component will not necessarily make use of OO technologies at all, but may instead provide a component based interface to an internal implementation based on alternative paradigms.

Components are further distinguished from the OO methodology by their design and intended use. OO classes are typically delivered as source code or one or more libraries, are commonly intended for white-box reuse, and a programmer is commonly required before their implementations can be properly understood and utilised. Meanwhile, components are commonly intended to be deployed and used as black box units of functionality. Component interfaces, disciplines, languages and standards also allow component applications to be deployed without an in depth knowledge of component implementation details.

Components embody concepts such as independence, late composition, and reuse by third parties, which are essential concepts for the design, marketing, sale and purchase of component implementations. In contrast, OO technologies are primarily concerned with implementation, and do not consider the deployment, assembly or distribution of objects beyond the compilation process. While the OO paradigm provides a range of concepts for the reuse of software implementations, CBSD delivers additional practices, standards and abstractions concerned with the development, distribution and deployment of computational abstractions as part of a software development methodology with larger scale abstractions and long term concerns.

Two concepts central to CBSD are software components, which are responsible for providing application functionality, and component composition, which is required in order to build more complex applications from collaborating component behaviour and is usually performed within the context of a particular component framework.

Components

The principal concept in CBSD is the binary software component, for which Szyperski [28] provides the following definition:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Software components represent the fundamental unit of composition from which component software is created. Each component is responsible for encapsulating a single coherent portion of functionality, and are independently developed, selected and deployed in order to collaboratively form a functioning software application. Components are units of *binary* composition, in that they do not define static (design-time) descriptions of behaviour, but instead encapsulate dynamic (run-time) behaviour. In the context of object-oriented programming, components may be seen as encapsulating objects rather than classes. Like objects, they may not be partially deployed or otherwise broken into smaller units of abstraction. Each component deployment is atomic, providing a well-defined unit of functionality, and more complex behaviour may only be obtained via their composition.

Not all component implementations are suitable for composition in all component applications: some components may have been developed for a particular platform or application. The functionality provided by some components may not be necessary or suitable for the application being developed, or they may require existing functionality that the application cannot provide. Some components may communicate using a different method or protocol to those supported by the application, and others may only be composed in a particular way. It is clear that only a subset of all components may be appropriate for use in any given application. More specifically, a given component application may only make use of those components that satisfy its *component model.*

Component Models

A given component model defines a particular set of component types, by taking into account their characteristic features, contractual obligations and environmental dependencies, communication styles, and how they may be structurally composed. Component models thus describe the most relevant aspects of component design that must be taken into account when deploying components as part of component based applications, which rely upon their collaborating functionality to provide the required runtime behaviour. A number of standards may be defined in order to describe those components that conform to the component model. For example:

Component Standards The term *binary software component* may be applied to a broad range of binary encapsulations at a variety of scales: components may range from single functions and procedures, to objects, libraries, and entire applications. The *component standard* for any given application context will be responsible for defining what is and is not a component within that context. A component standard refers to the collection of definitions, requirements and rules that must be adhered to in order for a component to function as part of a given application context.

An important aspect to consider when defining what components are allowed is their degree of encapsulation. The most common abstractions are black, white and gray box [29]. The functionality of black box components may only be accessed via their interfaces. Black box components are similar to OO objects, in that they do not reveal anything about their implementations. In contrast, the implementation details of white box components are fully visible to their clients. Gray box components offer partial access to select parts of their implementation details while hiding others. The exposed portions of a gray box component implementation may take the form of low level code, additional documentation, or even functional specifications describing internal behaviour.

Interface Standards Software components are specifically and explicitly designed for composition in order to form part of a (potentially) wide range of software applications. It is therefore important that each component also explicitly specifies its part in such compositions. This is most commonly achieved by describing the services the component provides, what the component requires from the environment in which it is deployed, and how communication takes place using *interface* or *wiring* standards.

Each component provides one or more interfaces, which are responsible for describing the services provided by the component. These interfaces are known as *provides interfaces.* It is via such provides interfaces that each component exposes the functionality it encapsulates to other components in the composition, and thus contributes to the collective functionality represented by the component software application.

Each component is also required to specify what it requires from such compositions. Such specifications are known as *context dependencies* or *requires interfaces*, and state the requirements that must be met before the component may form part of a given composition. A component's provides and requires interfaces are collectively known as the component's *bottleneck interface.*

Other details that may be available via component interfaces include version information, the description of additional options available to clients, and formal descriptions of implementation behaviour. However, the primary role of interface standards is to facilitate the specification of how component functionality may be interconnected to form component applications.

Composition Standards A component model will also specify a *composition standard*, which dictates the structural relationship between component deployments. The most common options for composition styles are *flat* compositions and their *hierarchical* counterparts. In the case of flat compositions, all components are deployed to the same global context, where they are identified, located and updated independently. By contrast, hierarchical compositions allow component deployments to encapsulate or own other deployments, leading to the hierarchical aggregation of components.

Component Frameworks

One aim of CBSD is to elevate the level of application development to a higher level of abstraction, where application developers may create applications via the independent selection, deployment, and composition of binary components. However, a number of lower level concerns must also be addressed by the resulting software.

For example, a component based application cannot entirely consist of component implementations alone, but must also embody, facilitate and enforce the various standards of a particular component model. A central process should be available to support the deployment of appropriate components, as well as their connection and runtime communication, while rejecting those components, connections and communications that do not satisfy the component model. Furthermore, the software may require access to lower level functionality, resources or facilities, such as those provided by the underlying operating system or distributed environment. It may not be desirable (or possible) for components and component developers to implement this low-level functionality.

Component frameworks offer a solution to lower level concerns by providing semi-complete realisations of domain specific software applications. Much like their object oriented counterparts described in Section 2.2.1, component frameworks are responsible for providing a stable yet adaptable basis for large-scale software development. This is achieved by incorporating the majority of application functionality and by including well defined locations and interfaces for application specific extension and customisation, as illustrated in Figure 2.8.

In the context of CBSD, component frameworks incorporate stability by implementing, supporting and enforcing a given component model. Meanwhile, individual component implementations and component compositions allow component software to adapt to a variety of applications. This combination of stable re-use and abstract adaptation can significantly improve the development of specific component software solutions.

A component framework will build upon a particular component and wiring standard in order to additionally define the deployment and communication standards by which components may be composed. A given component framework will thus support the instantiation and deployment of valid components (as specified by the component specification), as well as providing the means by which components may communicate via their specified interfaces. In the latter case, while a given

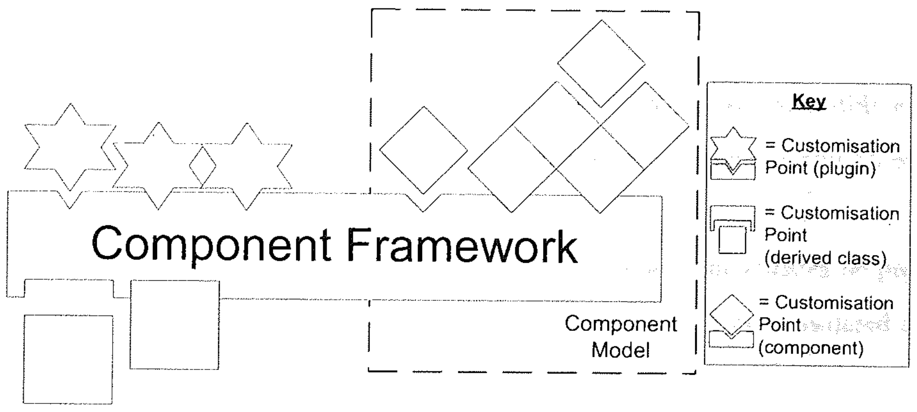

Component Framework
Component Model
Key
= Customisation Point (plugin)
= Customisation Point (derived class)
= Customisation Point (component)

Figure 2.8: An illustration of the customisation points provided by component frameworks.

component standard may state the required wiring standards by which components may communicate, it is the framework's responsibility to support such wiring standards via the provision of appropriate enumerations, data formats, marshalling services, and so on.

Composition Languages

Composition languages are used to describe component applications as the *concrete* composition of components. Composition descriptions typically drive the runtime deployment and subsequent connection of component instances within the context of a particular component model. In order to maintain consistency with the model, the composition language must also enforce that the components, connectors and compositions described adhere to underlying component, interface and composition standards. Composition languages provide an expressive tool by which software developers may describe the initial (and in some cases dynamic) deployment of computational elements, and their connections, to form component-based applications.

A considerable range of existing composition languages is available to the component software developer, with each language offering a different approach, paradigm and methodology for defining component based software. However, the majority of composition languages subscribe to some or all of the following requirements [30]: a composition language must support the encapsulation of both objects and components, it must support objects as processes, and components as abstractions. A composition language must also enforce *plug compatibility* between objects and components, provide a formal object model, and be scalable. The following paragraphs present a short description of each requirement in order to further define what a composition language is.

Composition languages introduce the requirement for the concept of a component. Component based methodologies can almost be regarded as analogous to traditional object oriented approaches, which similarly define software as a collection of independently operating elements connected via their interfaces. However, components provide an additional layer of abstraction above that of an object: whereas objects are computational elements representing runtime state and behaviour, components exist to define the composition of binary units via their interfaces. While the exact difference between objects and components will depend on a given component model, components and objects typically perform different roles as part of component based applications.

In the context of component based software, objects are independent entities or processes that collectively contribute to overall application functionality. Objects should be considered as operating autonomously and potentially concurrently, and should not be regarded as static data structures or collections of procedures. The objective of component based methodologies is to direct the collaboration of independent object behaviour to form higher level functionality that satisfies the requirements of component applications.

Components are software abstractions that must be instantiated and composed in order to become part of a component application. Components provide abstractions so that the composition of runtime functionality may take place. However, components do not necessarily correspond to objects; numerous concepts such as interfaces, wrappers, and connectors do not have corresponding object representations, and only have meaning within the context of a given composition. Certain concepts, such as connectors, are explicitly defined by composition languages and may be used to infer meaning from compositions. By contrast, many implementation methodologies will make such concepts implicit by hiding their use as implementation details (for example, consider the definition of inter-object references in object oriented approaches). Furthermore, the scale and granularity of components may have greater variation than that of objects in component based software. While the behaviour of all objects will contribute to overall application behaviour, some components may have to form part of other compositions before they become useful.

Component applications are formed by the composition of both objects and components. The composition of objects provides complex behaviour that defines application functionality, while the composition of components determines their communication and collaboration. The *plug compatibility* of objects and components must be formally expressed in order to form valid compositions, and to prevent invalid compositions from forming. Varying forms and degrees of plug compatibility exist. For example, object composition will typically require object interfaces and inter-object communication to adhere to an underlying type system and message passing protocol, while component compositions may take advantage of interface and parameter adaptation, message forwarding, and so on. The interface standard of a given component model is typically responsible for defining what plug compatibility means, and to what degree the composition language will support and enforce the composition of both objects and components.

Formally describing the roles of objects, components and compositions is sometimes considered a prerequisite to defining compositions representations and compositions languages [31]. A formal model can be used as a basis for enforcing a number of concepts, such as the integration and plug compatibility of objects and components, that are central to component based methodologies [32]. Subsequently, a variety of composition languages include a formal model of objects, components and compositions as a foundation for further descriptions. This foundation can be used to formalise the various standards of a component model, but it can also be used to define the enforcement of such standards as part of a component framework and supporting tools.

One further key requirement of composition languages is that they must also be scalable. Scalable in this context is not limited to size or granularity (which have already been discussed above), but also applies to configuration, completeness, correctness, and differing modes of use. When rapidly prototyping an application, a composition language must be flexible to allow dynamic changes, provide default values, and support incomplete or even incorrect definitions to improve productivity. By contrast, component software intended for delivery should be statically checked for correctness, completeness and validity. Perhaps a better term for this requirement would be *flexibility*: a composition language should be flexible to support the productive development of component applications as well as their correct deployment.

Composition languages provide the means by which component compositions may be defined. The use of declarative notations is common, in part due to their focus on the explicit deployment and connection of individual component instantiations. While composition languages often provide an expressive interface, they are also responsible for enforcing the rules and restrictions of the component model by limiting the compositions application developers may describe. As a result, composition languages will commonly support validation, contractual programming, and formal semantics. Although many existing composition languages subscribe to combinations of the requirements described above, a number of composition language developers are also experimenting with the use of alternative technologies such as XML [33], visual component assembly [34], or the integration of related methodologies such as aspect-oriented programming [35].

Existing Component Technologies

A number of existing component models and frameworks are available to application developers, in order to support the implementation of components and their deployment and composition as part of coherent applications. The most popular technologies are those developed by the Object Management Group (OMG), Sun, and Microsoft. The following is a brief overview of their respective contributions to the field of component based software development.

The Object Management Group developed the Common Object Request Broker Architecture (CORBA) standards in order to provide a uniform distributed computing environment for a wide range of programming languages and platforms. The CORBA standard defines mappings to and from its Interface Definition Language (IDL), which facilitates communication between different implementation languages. Furthermore, the same IDL is used to describe the interfaces to code implementations, known as CORBA servants, and thus determines the marshalling and unmarshalling required for code to interact via remote method invocation or Remote Procedure Call (RPC).

With CORBA 3, the OMG provides an extension similar to Enterprise Java Beans (EJB) in the form of its CORBA Component Model (CCM). A CCM application is an assembly of CCM or

EJB components, where CCM components consist of implementation code packaged alongside an XML description of the component's capabilities, attributes, and configuration.

Sun Java is a popular and widespread object-oriented programming language supporting a number of component models (specifically: J2EE, applets, JavaBeans, Enterprise Java Beans, servlets, and application client components).

The JavaBeans component model embodies Java's first component-based architecture. JavaBean components are called *beans* and consist of a number of classes and resources, although component instances are also called beans. One or more implemented beans are packaged as JAR files, which will also contain a manifest file that names the beans in the JAR file. During design time, an application designer will assemble JavaBean components to create component software compositions; at runtime, the required bean components are instantiated and together form an executing application.

Microsoft have two major standards for component based software development: COM, and the .NET framework. COM is the older of the two technologies, and while the underlying principles have been used elsewhere, COM is only really widely used on the Windows platform. COM is a binary wiring standard - it does not describe higher-level concepts such as objects, or components. Instead, the COM standard specifies the lower-level concept of an interface node, which may serve to represent (part of) a component.

.NET is the more recent of Microsoft's two component standards. It contains, among other things, the common language runtime (CLR), which is in turn an implementation of the common language infrastructure (CLI) specification. The CLI defines a language-neutral platform for software definition (via an intermediate language), deployment (using Assemblies), and extensible metadata. Assemblies are the .NET equivalent of software components. Similar to JAR files, Assemblies contain the executable implementation of the component's functionality, expressed as common intermediate language (CIL) byte code. Assemblies must also include a manifest, which describes the Assemblies' contents, as well as their dependencies.

In addition to the technologies described above, a wide range of composition languages have been developed as part of ongoing academic research. The following gives a brief overview of those languages most relevant to the material presented later in this work.

CoML [33] is an XML based composition language designed to be independent of the language(s) used to implement components. The CoML composition language forms part of a larger work targeting component construction, deployment, and assembly [36]. Due to its clear basis on

BML [37], CoML is most suitable for component platforms with similar functionality to JavaBeans, most notably the .NET framework.

VISSION [38] is a component-based dataflow framework for simulation and visualisation. VISSION's solutions are written entirely in C++, consisting of components deployed as C++ DLLs, and component compositions written in an interpreted form of C++. Due to use of C++ as a composition language, VISSION benefits from continuous support of C++ types, syntax and semantics from lower implementation levels through to higher levels of abstraction describing the composition of component applications.

Contigra [39] is an XML and eXtensible 3D (X3D) based architecture for component oriented 3D applications. Contigra's application descriptions include the instantiation and configuration of components at varying scales, and also includes a form of XML node inheritance that allows configuration prototypes to be reused and customised throughout such documents.

ChefMaster [40] is a glue framework which, unlike the majority of components frameworks discussed as part of this work, does not support a composition language of its own. Instead, the glue framework relies on a range of conversions from various composition language to ChefMaster scripts, which then drive the process of component composition. As part of this process, the grey box Enterprise JavaBeans components supported by ChefMaster may be modified by injecting behavioural adjustments, in order for such components to be used outside the contexts for which they were initially developed.

XCompose [41] is an XML based framework expressing complex component compositions as sequences of composition operators. Composition operators manipulate components in order to form composites, and may be glued together in order to form larger manipulations sequences called *composition pattern templates*, which closely resemble imperative functions.

PICCOLA [26] is an imperative glue code, derived from $\pi\lambda$-calculus, conforming to the functional programming paradigm. Component applications are assembled from cooperating components, or *agents*, which communicate by exchanging *forms*. PICCOLA's forms are extensible records, constituting a fundamental concept used for deployment as well as message passing; forms thus have a widespread use similar to tables in Lua.

VHD++ [42] is a component framework supporting interactive, real-time 3D simulations targeting game, virtual reality and augmented reality domains. VHD++ is a feature-rich component framework that has been successfully applied in a range of areas including education, training, and construction. The VHD++ framework also makes use of an expressive XML based composition language, includes support for behavioural scripts written in the Python language, and exposes a number of customisation points at various levels of abstraction for future customisation and extensibility.

Component based approaches are becoming more commonplace in contemporary software solutions and applications. However, it is clear that today's CBSD is an immature and developing methodology: certain CBSD proponents have recently highlighted the limitations of current component technologies [43, 44]. This viewpoint is further supported by a number of ongoing studies, which continue their attempts to resolve a broad range of problems and shortcomings.

Current CBSD lacks the support of methodologies such as RUP [23], models such as the waterfall and spiral models [45, 46], or standards such as UML. Consequently, many component software developers are developing systems from scratch, thereby failing to take advantage of the reusability, maintainability, and evolving development of component based solutions [47]:

> Components are mostly identified in the late phases of the system development cycle without considering the end-users' requirements specified in the early phases. The effort required to develop or re-use components which satisfy the requirements is still significant, so that a lot of developers generally prefer to develop a system from scratch, while being largely influenced by technological concerns.

Furthermore, component based software commonly fails to properly separate application functionality into loosely coupled, coherent component implementations: code is often duplicated across many components, while business logic may be tangled with code specifically concerned with the use of a given component platform. Current research aims to solve this problem by applying aspect or attribute oriented approaches [48, 49].

However, it is likely that a higher level, further reaching solution may be found in the application of CBSD methodologies, models and standards. Consequently, a number of studies aim to introduce software development methodologies for component based software development [47], component frameworks [50, 51], and component classification [52]. A subset of this literature is also dedicated to resolving incompatibilities between middleware platforms [53] and component versions [54]; certain researchers suggest a combination of CBSD and model driven approaches [55, 56].

Although the CBSD research community continues to propose new component technologies and methodologies, much of the literature is dedicated to resolving gaps and inadequacies in current technologies. This is particularly true for the current limitations of component description, use, and *integration*, perhaps due to an ever growing desire to make use of existing component implementations. Recent research focuses on adequately describing and measuring quality of service [57, 58] and non-functional requirements [59], while further work looks at the quality, certification and trustworthiness of component implementations [60, 61, 62]. Such research indicates that binary components are not enough, and that there is an increasing demand for interface description languages (IDLs) to incorporate additional descriptions, features, and functionality.

Finally, it is becoming clear that component based solutions may not be appropriate for all problem domains due to limited runtime performance. In particular, it has been shown that CBSD may not be suitable for high-performance systems [63], while its applicability in embedded systems could be limited by current component technologies [64]. However, this is not a universally held view, and component based research continues to explore applications in a diverse range of domains. For example, the successful use of component based approaches in computationally intensive applications is demonstrated by modern data driven game technologies, as discussed in Section 2.3.1. Such games satisfy conflicting requirements for software designs that are flexible to change at a late stage in the software development process, and the efficient runtime performance of the designs' corresponding implementations. An effective balance is achieved by encapsulating the majority of computationally intensive software as low level software libraries, while dedicated notations closely related to component approaches facilitate title specific customisation at a high level of abstraction. It can be speculated that a broader application of such game development techniques may have potential benefit to more mainstream computer science CBSD applications, particularly where runtime performance is important.

While the performance of complex component applications can be greatly influenced by early design decisions [65], developments in the measurement and prediction of performance for component software may lead to more efficient designs, and thus more successful use [64]. In the meantime, operating systems and other user driven applications continue to use component based approaches with much success.

Despite a number of potential limitations, CBSD improves upon the OO programming paradigm in order to provide an approach that encompasses a broader view of the software development lifecycle. At the heart of CBSD lies the binary component, which represents a range of concepts that are typically more coarsely grained and abstract than those seen in OO methodologies. As with interpreted languages, component based approaches thus take a step further from the hardware platform and toward a higher level of abstraction: CBSD incorporates a design phase based on the selection, interconnection, and configuration of high level abstractions; the initial development and subsequent reuse of implementations may involve the use of a component market, while software maintenance is facilitated through the individual update of versioned components. Although component implementations necessarily remain concrete and platform dependent, a wide range of higher level concepts are available to describe overall application behaviour and support the component based methodology as a whole.

Architecture description languages continue the movement into higher levels of abstraction, often building upon concepts established by component based approaches, by focusing on the description, analysis and reasoning of software architectures.

2.2.4 Architecture Description Languages

A software architecture is the top level decomposition of a system into major components together with a characterisation of how these components interact [66]. A software architecture is a global, often graphic, representation suitable for communication among customers, stakeholders, designers and software engineers. Software architecture is a high level representation derived from software requirements, and as such, captures early design decisions by explicitly defining software functionality as interactions between computational components. Software architecture provides the first system based codification of software requirements, and directs subsequent activities such as structural design and software implementation. Because of their early involvement in the software development process, the designs represented by software architectures are the most difficult to change. There is a growing interest in being able to reason about architectural designs, as well as testing the feasibility of design candidates, before they can impact later software development activities.

Architectural Description Languages (ADLs) are emerging as formal notations for representing and reasoning about software architectures. The aim of ADLs is to enhance the understandability and reusability of architectural designs, and to enable a greater degree of analysis. Due to the varying features and functionality of ADLs described in the literature, it has been difficult to define exactly what an ADL is, and conversely, what an ADL is not. As an early step toward producing an appropriate definition, a number of ADL proponents have presented classifications that identify ADLs based on their capabilities [67, 68, 69]. The following definition includes a number of widely accepted requirements [68]:

> An ADL must explicitly model *components*, *connectors*, and their *configurations*[4]; furthermore, to be truly usable and useful, it must provide *tool support* for architecture-based development and evolution.

Following the typical contents of ADL classifications, the following description focusses on the representation of *components*, *protocols*, *connectors*, *configurations* and *tools*.

Components in ADLs represent the first description of software functionality. ADL components are therefore typically coarse grained, consisting of a number of behaviour patterns and responsibilities, which may be as large as applications. However, some ADLs can also incorporate concrete behaviour in the form of single procedures.

[4]In the context of this definition, *configuration* refers the topology of components and connectors.

Unlike composition languages, ADLs are also concerned with the specification of component behavioural semantics. Such specifications will typically include a (formal) notation for describing functional and non-functional aspects of component behaviour, as well as usage requirements and constraints. While the functional behaviour of a component refers to the computation it encapsulates, the non-functional aspects of a component incorporate a number of characteristics such as safety, security, reliability and performance.

Protocols describe the various ways in which components communicate, including restrictions on which components are permitted to communicate. Certain ADLs support the specification of communication protocols as part of individual component interfaces. In such cases, a component interface will typically define what data types are communicated as inputs and outputs for its exported functions and procedures. Other examples of protocol descriptions include the stipulation of communication styles (including synchronous and asynchronous message passing) and additional restrictions concerning valid inter component connections.

Connectors are modelled explicitly in the majority of ADLs; they are typically defined either in-line as part of component configurations, or as independent entities described separately. ADL connectors are used to model the numerous interactions and styles of interaction among components, and will not necessarily correspond to actual objects or components in lower level software representations. As with components themselves, the level of abstraction and granularity supported by connectors varies greatly among ADLs. In some cases, connectors consist of inter-component references, and provide in-line topological information only. In other ADLs, component connectors are distinct entities, sometimes forming their own type families for enhanced reuse, and may incorporate their own imported and exported services, specifications for behavioural semantics, communication protocols, and so on.

Configurations in ADLs refer to the topology of components and connectors defining a particular software architecture. A vital requirement of ADL configurations is that they can be created, manipulated, read and understood by humans, so that software architectures may be effectively described and communicated. However, it is also important that ADL configurations are open to manipulation by tools. Furthermore, an ADL configuration must be suitable for refinement to more concrete levels of abstraction, including structural designs and concrete implementations in a given programming language.

Tools have a number of applications within the context of ADLs. For example, tools are commonly employed during the processes of defining, viewing, or otherwise manipulating software architecture configurations. ADL tools may also be used for the analysis of existing configurations.

For example, the validity or feasibility of a given architecture may be determined by taking the requirements and restrictions of components and connectors into account. Similarly, a tool could indicate which components invalidate an architecture's non-functional requirements by inspecting their behavioural specifications. A further use of ADL tools is to provide mappings from the abstract representations supported by ADLs to more concrete representations at lower levels of abstraction, employing code generation, scripting, and glue.

While the classifications presented in relevant literature present the most common requirements for ADL capabilities, the provision of such capabilities varies greatly among ADL implementations. A more general description of ADLs is that they typically provide similar functionality to composition languages, albeit at a higher level of abstraction. The need to reason about and analyse architectural designs leads to the incorporation of additional features such as explicit modelling of connectors and behavioural specifications, which are not required for lower level structural designs intended to inform software implementation. ADLs are intended to encapsulate the high level architectural elements of software design, and embody the first system based codification of software requirements. ADLs are formal notations for the description, communication and both functional and non-functional analysis of software architectures.

Despite the clear distinction between the requirements of ADLs and composition languages, they exist on the same continuous scale of descriptions for software composition. The diverse range of ADL and composition language implementations presented in the literature has lead to a blurring of their individual roles, as ADLs include a variety of lower level definitions, while composition languages incorporate a growing number of higher level abstractions. Furthermore, a number of papers present efforts to explicitly bridge the gap between the two technologies [70, 71, 72].

While technologies at the lower level of component implementations and component frameworks are currently being applied in industry, most notably via a small number of increasingly popular technologies, the use of ADLs to specify component based software architecture remains largely academic. The immature nature of ADLs is demonstrated by a wide variety of continuing academic research, including fundamental issues such as the semantics of component connectors [73], as well as higher level concepts such as the redeployment and reconfiguration of component architectures [72].

As with lower level component based approaches, ADLs currently lack a unified standard for the development of software implementations and applications based on reasoning at the level of software architectures. A number of methods have been proposed for bridging the gap between abstract, architectural level components and concrete implementation level components [70, 74]. However, until a common standard is widely supported, the lack of guidance when utilising ADLs is likely to continue to discourage commercial applications, leading to ad-hoc and bespoke implementations [75].

The lack of standards for ADLs is possibly due to the immaturity of ADL technologies, as today's ADLs continue to explore the overall role and use of architectural descriptions as part of software development. Furthermore, ADL technologies have yet to converge, with support for architecture configuration, refinement and traceability, evolution, dynamism, and non-functional properties varying across the available languages [76]. The diverse nature of ADLs may be partly due to each ADL being designed with a single particular goal in mind; for instance [69]:

> Darwin is designed for dynamic architectures, Unicon is aimed for generating executable code from a description, ACME focusses on architectural interchange, and Rapide on simulation and architecture conformance. Wright precisely focusses on the integration of formal methodology to architectural description.

Component based software development is a developing and immature field, with an assorted range of open issues to be resolved by collaborating industrial and academic research and development [44]. Consequently, today's component based applications employ a variety of experimental and proprietary technologies, which build upon popular standards from a small number of vendors. The resulting collection of component definitions, models, frameworks, and languages is overwhelmingly diverse, although a future convergence and standardisation of technologies may further their acceptance and industrial application.

ADLs provide a formal representation of software architectures. While the conceptual elements supported by ADLs are typically both more abstract and more granular than those supported by component based approaches, the distinction between the two technologies is becoming increasing blurred. ADL proponents make use of dedicated tools that permit early reasoning and analysis of non-functional aspects of software architectures. While some ADLs also allow their high level descriptions to be translated to lower level representations, the process of automatically generating concrete implementations from higher level notations is the particular focus of model driven approaches to software development.

2.2.5 Model Driven Software Development

Model driven software development (MDSD) focuses on the definition of high level models, which encompass software design, and the subsequent automatic transformation of such models to generate their corresponding software implementation. In this way, MDSD can be considered similar to computer aided manufacture, where blueprints are transformed into physical products by automated machines and tools. The goals of MDSD include increased development speed and enhanced software quality, both of which are enabled by the use of an automated code generation process. By generating low level implementation details from more abstract descriptions, software code can be produced automatically, quickly, and consistently.

MDSD relies upon the availability of domain specific languages, which are used to describe an abstract, platform independent model (PIM) of the desired software application. MDSD also makes use of a variety of languages that define mappings, or transformations, from higher level models to increasingly platform specific models (PSM), ultimately including software code in a particular programming language. A range of tools facilitate the process of producing initial high level models, transforming from higher to lower level models, and finally editing, deploying and executing the generated software code. An overview of the MDSD process is presented in Figure 2.9; further detail follows.

Models In many software development methodologies, modelling forms part of an initial design phase. The software model is typically used to inform software developers during a subsequent implementation phase, after which the model is no longer necessary. By contrast, MDSD empowers the software model to include all information that is required to automatically generate its corresponding implementation; the software model constitutes an important part of the software's definition. To this end, MDSD models must be both abstract *and* formal at the same time. While a high level of abstraction facilitates expressive software design, formal elements are required in order to enforce the rules of a particular problem domain. Because of the wide range of MDSD applications, today's modelling notations are typically domain specific.

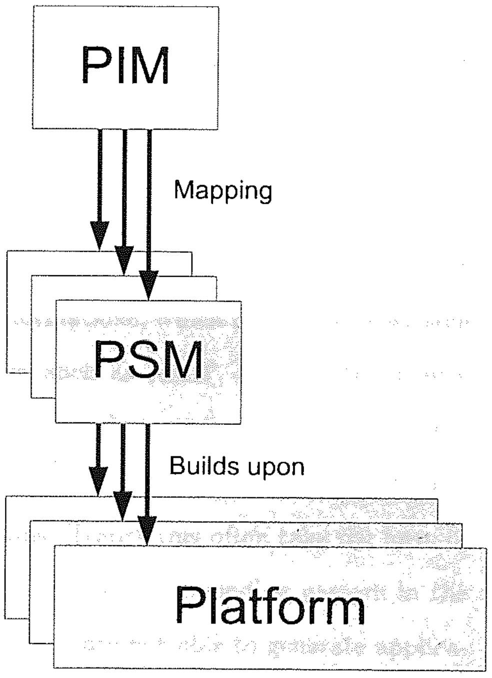


Figure 2.9: An overview of the MDSD approach, illustrating the mappings from platform independent models (PIM) to more platform specific models (PSM).

The MDSD methodology begins with the development of an initial model. The model will embody an abstract, high level view of the application's design, incorporating concepts that are independent of any particular application platform. The initial model thus abstracts from implementation or technological details, and instead focuses on those stable concepts that are core to the software design. For example, UML static class diagrams, sequence diagrams and activity diagrams may be used to depict a high level view of a given software design [77].

Platforms In MDSD, the concept of a platform is not limited to a particular hardware environment or operating system, but encompasses many levels of abstraction: hardware environments are a platform for operating systems, which provide platforms for programming languages, which are in turn platforms for software applications, and so on. The concept of a target platform is introduced during the process of mapping from high level models to lower level, platform specific, models that must take platform differences into account. Model *transformations* are responsible for introducing successive platform specific elements in order to translate a more general model into a more specific one, within the context of a particular platform. While higher level models commonly consist of graphical or other abstract notations, lower level models will typically introduce code skeletons and abstract interfaces, then specific patterns, data structures, algorithms, and finally concrete code for functions, procedures and class methods.

An MDSD platform will commonly incorporate a software framework or selection of components ready to be assembled. In such cases, the MDSD approach will not be required to produce low level code providing all software behaviour, but will instead reuse existing implementations in order to provide the required application functionality.

Transformations Each translation of an element in a higher level model to its corresponding element in a lower level model is performed by one or more transformations, which are typically written in languages ranging from dedicated transformation languages such as XSLT, to mainstream programming and scripting languages such as C++ or Python. Example transformations include the generation of class definitions from models such as UML static class diagrams, and the generation of lower level algorithms from UML activity and sequence diagrams. Transforms often take the form of *templates*, which replace a specific pattern in the source model with a corresponding pattern in the destination model. However, in many cases MDSD transformations are not able to generate application software in its entirety, and software developers may be required to provide the more complex implementations.

Building upon existing specification, selection, integration and implementation technologies, model driven approaches automatically derive software solutions from high level notations and descriptions. In this way, MDSD provides a top-down development methodology with increased development speed, software quality and consistency. Although there is a limit to what can be automatically generated from higher level models, improvements to independent software development technologies, particularly in the areas of formal notations and standardisation, may allow for more complex solutions to be written by MDSD transformations.

The top down approach embodied by MDSD has clear links to the use of composition languages and

ADLs to drive the selection and interconnection of platform dependent computational encapsulations. Furthermore, component based approaches may be combined with MDSD, with higher level notations driving the assembly of lower level component implementations [55].

2.2.6 Summary of Software Development Methodologies

Object oriented methodologies provide a popular approach for the definition of software implementations, while also defining a fine grained mechanism for their encapsulation and reuse. Meanwhile, software libraries and frameworks supply similar modularity at higher levels of granularity and abstraction, in order to facilitate the incremental development of applications.

The OO paradigm is widely supported by implementation languages, where the expressiveness of OO abstractions may be combined with a range of low level concepts with a close correspondence with the underlying execution platform. Consequently, programmers making use of OO implementation languages are able to write OO classes, libraries and frameworks, but must also consider details such as heap memory management, thread synchronisation, and the availability of platform specific resources such as hardware timers. Programs written in implementation languages are typically compiled to platform specific machine instructions, and the resulting binary representations are commonly associated with high runtime performance and efficient memory use and execution.

However, the lower level programming environments provided by implementation languages can lead to less expressive notations than those supported by their higher level counterparts, leading to programs with more instructions and hence lower programmer productivity. Furthermore, the close correspondence between implementation languages and platform hardware results in implementations being less portable and flexible. Low level solutions may also be less adaptive to future change, and may be difficult to integrate with independent implementations.

Interpreted languages offer a higher level, more expressive software development environment than implementation languages, in part due to the availability of a dedicated virtual machine (VM), which provides a foundation of basic functionality that interpreted languages can rely on. Interpreted languages also commonly offer a range of abstract concepts or programming paradigms, often making them more productive than implementation languages, and thus more suited to rapid application development. For example, Lua's tables are a fundamental and yet incredibly powerful part of the language providing a range of data structure concepts in the form of a single abstraction. Tables in Lua take the form of associative containers, although the values on the left and right hand sides of the association can be any of the concepts supported by the Lua language. For instance, tables support integer based indexing in order to support array behaviour, direct manipulation of the first and last elements can also be used to provide list, queue and stack-like data structures. Indexing using strings allows items to be stored and subsequently accessed by name. However, string based indexing also forms the basis of program structuring, as further tables, functions, procedures and items can all be stored as part of a named table hierarchy. The provision of a single, high level concept to support a wide range of data structures allows Lua programs to be written more rapidly and expressively than programs written in lower languages such as C++, which has a number of data structures available and is optimised for different uses.

Unlike their implementation language counterparts, programs written using interpreted languages are typically compiled to an intermediate bytecode for later interpretation by a VM. As compiled bytecode can be interpreted by an appropriate VM on any execution platform, programs written using interpreted languages are more portable than those written using implementation languages. Furthermore, although the overhead of VM interpretation has been prohibitive for larger applications in the past, improvements in compiler and VM technologies, as well as continuing advances in CPU hardware, are effectively mitigating the negative aspects of interpreted languages. Some interpreted languages, such as Lua and Python, were also designed to support a range of bindings to lower level languages including C and C++, so that existing implementations may be integrated and exposed to programs written in higher level languages.

Implementation languages are typically used for low level programming, and interpreted languages for writing software at higher levels of abstraction, often relying on libraries written using implementation languages for those parts requiring more efficient execution. In contrast, component based approaches focus on the *assembly* of independently developed implementations in order to form complete applications. CBSD builds upon the interests of lower level technologies, and is primarily concerned with the definition, selection, interconnection, configuration, and larger scale distribution of components. To this end, the component based methodology introduces a range of technologies that collectively support and enforce a new way of designing, developing, describing, and distributing software encapsulations and applications.

In a similar way to interpreted languages, a composition based approach may be suited to rapid application development, provided the correct components are available for purchase and existing technologies are being used to support their composition. Furthermore, CBSD is particularly suited to the development of a range of related products, where the use of component technologies will emphasise the reuse of implementations. Due to the use of high level composition languages to describe component based applications, component based solutions may also be more flexible to change and open to high level analysis than solutions written using lower level languages.

At a higher level of abstraction, ADLs focus on the specification of software architecture, often incorporating measures related to non-functional properties and quality of service provided by the runtime execution of component based applications. While a wide variety of features are present in lower level composition languages, ADLs are commonly concerned with modelling, manipulating, analysing and ultimately understanding software architectures.

Today's vast range of component based technologies present a variety of benefits. Common themes include greater flexibility and extensibility, enhanced implementation reuse, and additional support for analysis. CBSD promotes a more effective methodology for designing, developing and marketing software implementations. However, the vast majority of component based technologies have yet to mature. Although a small number of solutions, including COM, CORBA, JavaBeans and .NET are becoming increasingly popular and more widely applied, such solutions form *part* of the component based approach. Composition languages, component frameworks, and ADLs remain predominantly academic. Furthermore, research in CBSD has yet to converge in order to provide widely supported standards for the design and implementation of component based implementations and applications.

Model driven approaches endeavour to provide an *automated transformation* from a number of abstract models to their corresponding low level platforms and concrete implementations. As part of this process, MDSD may make use of a wide range of technologies providing sequential mappings from platform independent models to increasingly concrete representations. Such technologies may include implementation, interpreted, composition and architecture description languages providing a wide range of increasingly abstract concepts, environments and encapsulations for software development, with each technology introducing additional abstractions that may be translated to lower level operations and representations. While such technologies may be used in combination in order to provide a complete software solution, MDSD may be used to bring them together as part of a continuous and automatic software development process.

Model driven approaches are not yet completely effective, and may not be able to provide a complete concrete implementation for all application designs. Where concrete implementations are not automatically derived, a software engineer may still be required to provide lower level designs and software code. However, a range of new functionality forming part of the new UML 2.0 standard give explicit support for the MDSD approach, and vastly improve the expressiveness and precision of modelling notations [77]. It is likely that the new features of the UML 2.0 standard will improve the effectiveness of model driven approaches, resolve a number of current restrictions and allow wider range of software solutions to be generated automatically.

The runtime behaviour of DDDAS is driven by high level, domain specific elements embedded in runtime data. Section 2.2 has discussed a number of technologies, with increasing levels of abstraction, where higher level descriptions of software may be used to drive lower level software structure and overall application functionality.

- The class descriptions of OO implementation languages allow the state and behavioural aspects of software to be described separately. Features such as inheritance and implementation encapsulation allow a degree of flexibility, but the low level nature of such languages is prohibitive to truly adaptive behaviour.

- At the opposite extreme, interpreted languages *focus* on providing software flexibility, and commonly support runtime adaptation via the dynamic generation and interpretation of plaintext scripts. However, such languages may be *too* flexible for the development of large-scale adaptive applications, where additional granularity may benefit the dynamic (re)deployment of independent software implementations.

- Component based approaches support an ideal separation of software implementation and deployment, and thus provide an opportunity to adapt software (re)composition at runtime from a collection of black-box implementations.

- Meanwhile, both ADLs and model driven approaches provide static software descriptions at higher levels of abstraction. While future research may allow ADLs and MDSD to be applied to DDDAS, current progress remains focussed on academic projects with limited industrial application.

Applying CBSD within the field of DDDAS has the potential to enhance the dynamic adaptability of future software systems. A component-based DDDAS would be able to include additional functionality at runtime, optionally replacing existing behaviour with alternative implementations, in order to optimise runtime behaviour in response to dynamically changing requirements.

While CBSD could be used to support the macro-scale manipulation of data driven behaviour, future DDDAS could incorporate greater flexibility if the component implementations themselves could also be data driven. However, the extensive use of data driven methodologies as defined in Section 2.1.2 is not commonly found in industrial or academic applications. As previously discussed, the term *data driven* typically refers to an emphasis on data processing, or that limited aspects of the software's behaviour may be parameterised using an appropriate configuration file.

Meanwhile, the computer games industry has adopted a particularly effective data driven approach to software development. Indeed, the popularity of data driven rendering has led to hardware vendors producing a range of data driven technologies, with a dramatic effect on subsequent software flexibility. The following section describes the data driven approach used by today's computer game developers. The aim of Section 2.3 is to provide an overview of the data driven programming paradigm, while motivating the combination of CBSD and data driven approaches in order to enhance software flexibility and adaptability.

2.3 Computer Games Technology

Superficially, the computer games industry appears to have little influence on computer science academia, and vice-versa: indeed, the two fields often appear to be independent. Under closer inspection, it becomes apparent that a number of significant computer game technologies are supported by continuing efforts from academic research. For example, popular methods for introducing realistic shadows to scenes depicted in games [78, 79] have clear academic foundations [80, 81, 82, 83]. Furthermore, computer games are often included as a motivation for a range of research efforts focussing on the interactive visualisation of dynamic scenes [84, 85]. Certain fields, such as those concerning virtual reality, incorporate a range of application content, interaction, and visualisation technologies whose application appears to be very similar to that seen in games [12]. Finally, computer game applications have also been employed to provide interactive visualisation capabilities for a range of research projects [86, 87, 88]. In such areas, the distinction between computer game technology and academic research is becoming less clear.

During the past decade, the game development community has made considerable efforts to improve its software development methodologies and technologies in response to a growing consumer demand for more detailed, realistic, interactive, immersive and customisable experiences. Best practice is consequently the focus of a wide range of game development articles and presentations [89, 90, 91].

In many cases, a modern game will consist of a number of distinct layers as seen in Figure 2.10: a lower level layer (left) providing generic functionality for a range of related titles; a middle layer (center) representing the functionality related to a single specific game title; and a higher level layer (right) allowing third parties or consumers, many of whom are non-programmers, to customise a specific game title to their own requirements. The lower and middle layers will often be implemented as a combination of first and third party libraries encapsulating tightly coupled responsibilities such as resource access and management, visual and audio output, user input, and runtime updates. These libraries form a software framework, commonly referred to as a *game engine*, providing a reusable backbone of functionality. In order to create a range of individual game titles, game developers will supply custom subclasses, plugins, and parameterisations in order to customise the game engine's operation according to title specific requirements, latest hardware capabilities, consumer market, and so on. Finally, a diverse range of title-specific environments and experiences are provided by a variety of content and logic data files.

A growing number of game engines have begun to incorporate a data driven aspect, as described in Section 2.1.2, in order to maximise the reuse of implementations, as well as their flexibility and extensibility. The most noteworthy applications of data driven methodologies in the context of computer games technology can be seen in the definition of runtime game content as described in Section 2.3.1,

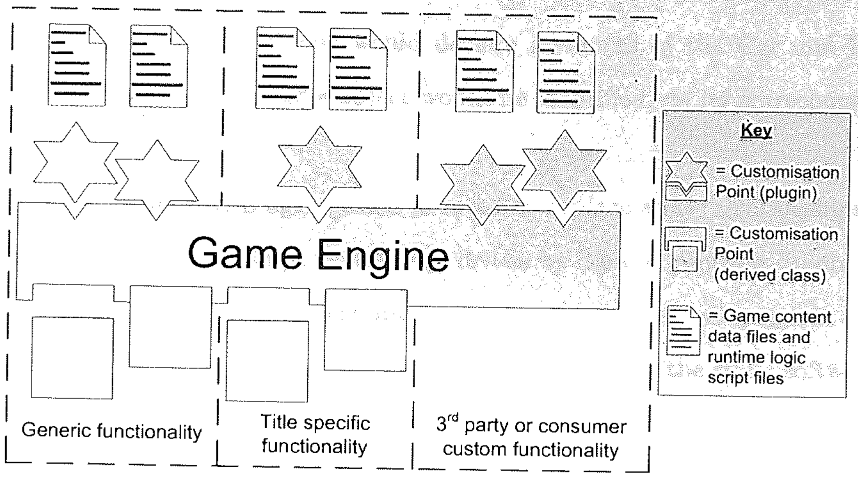

Game Engine
Generic functionality
Title specific functionality
3rd party or consumer custom functionality
Key
= Customisation Point (plugin)
= Customisation Point (derived class)
= Game content data files and runtime logic script files

Figure 2.10: An overview of the architecture of a typical computer game application.

and scripted behaviour in programmable rendering pipelines as outlined in Section 2.3.2.

2.3.1 Object Oriented Data Driven Programming

As part of the movement toward more content driven methodologies, games developers are making increasing use of data driven programming (DDP), as described in Section 2.1.2. Game developers extend the popular data driven concept by incorporating object oriented features such as type definition, inheritance and instantiation, and enhance the use of DDP by allowing application data to *drive runtime software structure*. In order to distinguish the game developers' extensions to data driven programming from that described in Section 2.1.2, the approach utilised by game developers will be referred to as *object oriented data driven programming* (OODDP) in the text that follows.

In describing his motivations for developing OODDP, Scott Bilas states [92]:

> To meet changing design needs, one can't just data-drive the object properties, one must data-drive the structure (schema) of the objects.

The term *schema* here refers to the objects, the objects' attributes, and the relationships between objects. In OODDP, this schema may be data-driven by replacing static inheritance relationships with dynamic aggregation relationships and then driving dynamic assembly at runtime using data.

For example, consider the following simplified object oriented software structure, as illustrated by Figure 2.11. A Vehicle type forms the root of an object oriented inheritance hierarchy, providing common behaviour (e.g. the ability to belong to a traffic network simulation, and to travel between locations on that network), and common attributes (such as engine size, average speed and so on). The PassengerVehicle type extends the Vehicle type by providing seats for passengers, plus the behaviour for boarding and alighting. The Car type extends PassengerVehicle in order to provide a concrete type for cars and similar vehicles, while the LightAircraft type provides a PassengerVehicle that can transport passengers by air. If this scenario were implemented in code, runtime objects would be instances of the child classes - that is, one would declare instances of the Car and LightAircraft types as part of the code, and each runtime object would be identified via its corresponding variable declaration.

OODDP replaces inheritance with aggregation in order to replace static relationships in the software structure with dynamic relationships that can be driven by data at runtime. In order to achieve this, related classes in the inheritance hierarchy must be represented using independent classes, which are then combined to provide the functionality previously represented by the static software structure. For example, the PassengerVehicle class no longer inherits from Vehicle, but is written as an independent class. If the parent (Vehicle) class from the earlier example provides any common behaviour to its children, then it will also be represented by an independent class that encapsulates this commonality. If the OODDP scenario were implemented in code, runtime objects would be represented by simple containers of instances of the independent child class types. For example, an instance of the Car class would be represented in the OODDP scenario as the aggregation shown in Figure 2.12, incorporating the functionality of the Vehicle class, plus that provided by the SeatingCapacity class. Each OODDP instance (that is, the container instance representing the corresponding class instance) is uniquely identified via a runtime identifier. The instance in Figure 2.12 has the identifier 'myCar'.

In OODDP, each hierarchy of container object and contained class instances may be described using an OODDP *type description*. Type descriptions drive the runtime instantiation and aggregation of container and class instances in order to form an OODDP hierarchy. Each type description includes an identifier for the OODDP type currently being described, and specifies which classes should be instantiated and inserted into the container. Each specified class instance is also given an identifier so that the various parts of an OODDP hierarchy may be uniquely identified. Class instance specifications may also include an optional configuration, which is used to customise the class instance and can be regarded as synonymous with class constructor parameterisation. In practice, class configurations are used to initialise attribute values.

Figure 2.13, written in XML, describes the type PassengerVehicleType, which contains a Vehicle class instance and a SeatingCapacity class instance, both of which include configuration elements.

Furthering its incorporation of the object-oriented paradigm, OODDP re-introduces inheritance

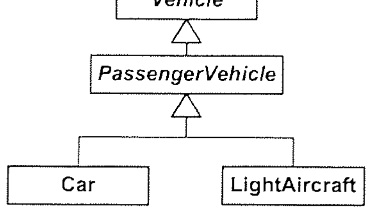


Figure 2.11: A simplified object oriented software structure.

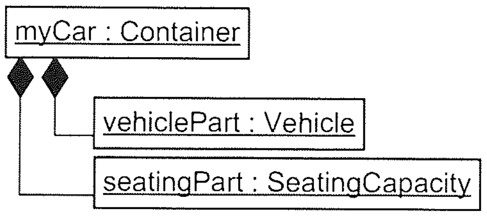


Figure 2.12: The PassegengerVehicle type is represented as an OODDP hierarchy of container and independent class instances.

```
<Type id="PassengerVehicleType">
 <Vehicle id="vehiclePart">
  <!-- Vehicle configuration -->
  <AverageSpeed>50mph</AverageSpeed>
  <!-- etc -->
 </Vehicle>
 <SeatingCapacity id="seatingPart">
  <!-- SeatingCapacity configuration -->
  <Seats>5</Seats>
  <!-- etc -->
 </SeatingCapacity>
</Type>
```

Figure 2.13: An example OODDP type description for the OODDP hierarchy from Figure 2.12. Vehicle and SeatingCapacity are the names of OODDP components, which will be added to all PassengerVehicleType instances.

via manipulations on its type descriptions. New OODDP type descriptions can be described based on existing OODDP type descriptions, allowing for description re-use and thus more expressive power for OODDP software designers. In a similar way to traditional object-oriented paradigms, OODDP inheritance forms a relationship between a parent and child type description. For example, Figure 2.14 makes use of a parent type description (PassengerVehicleType from Figure 2.13) and describes a child type description (LightAircraftType). The LightAircraft type describes PassengerVehicleType type that uses flight as its mode of travel, as represented by its additional FixedWingFlight class instance.

An OODDP type description may be instantiated via the definition of an OODDP *instance description*. An instance description typically has a similar appearance to an OODDP type description, and provides an identifier for the instance. The following excerpt describes an instance of the Passen-

```
<Type id="LightAircraftType" parent="PassengerVehicleType">
 <!-- LightAircraftType inherits seatingPart from the PassengerVehicleType    -->
 <!-- OODDP type description. LightAircraftType provides its own description -->
 <!-- for the vehiclePart, which overrides that in PassengerVehicleType      -->
 <Vehicle id="vehiclePart">
  <AverageSpeed>150mph</AverageSpeed>
 </Vehicle>
 <!-- LightAircraftType extends PassengerVehicleType by including an -->
 <!-- instance of the FixedWingFlight class                           -->
 <FixedWingFlight id="flightPart">
  <Ceiling>15000ft</Ceiling>
 </FixedWingFlight>
</Type>
```

Figure 2.14: An OODDP type description for the LightAircraftType type, which makes use of OODDP inheritance to both include and extend the definition of the PassengerVehicleType type description given in Figure 2.13.

gerVehicleType type given in Figure 2.13:

```
<Instance id="myCar"parent="PassengerVehicleType"/>
```

After OODDP inheritance is applied, the 'myCar' instance description will include the contents of the PassengerVehicleType type description as given in Figure 2.13.

OODDP inheritance, as described above, may also be applied to instance descriptions. For example, the following describes another instance of the PassengerVehicleType type given in Figure 2.13, although in this example the instance description overrides the PassengerVehicleType type description by providing a custom SeatingCapacity class description for a custom number of seats:

```
<Instance id="anotherCar" parent="PassengerVehicleType">
 <SeatingCapacity id="seatingPart">
  <Seats>7</Seats>
 </SeatingCapacity>
</Instance>
```

While the basic principles of OODDP are commonly accepted as described above, their implementation often varies due to individual developer requirements. Alex Duran [93] presents a range of data driven programming technologies, providing a clear overview of common features among data driven systems. In addition, the following proponents' implementations are particularly relevant to the concepts presented later in this work.

Scott Bilas [92] Introduces an OODDP system incorporating types, instances and inheritance relationships, which are described and configured using a dedicated scripting language. Bilas' system makes use of configuration schema, which perform a similar role to XML schema, in order to both describe and enforce the structure of OODDP types. Components can be developed independently as C++ libraries or as scripts, and may be introduced to the OODDP system to form part of its type and instance descriptions. Bilas' system also makes use of automated function binding capable of forming links with the functionality exposed by C++ DLLs [94].

Chris Stoy [95] describes a system based on `GameObjects` representing concepts such as player avatars, with each `GameObject` consisting of `GameObjectComponents` encapsulating a range of finely grained concepts. Stoy's system has support for data-based configuration templates, which can be used to initialise `GameObjects` and `GameObjectComponents` during the data driven instantiation of a given game scene.

Bjarne Rene [96] presents a more flexible component communication system based on message passing. Although not explicitly demonstrated by the literature, it is clear that Rene's system can be used to develop independent, loosely coupled components that do not rely on other components' implementations for inter-component communication.

Modern game titles are becoming increasingly data driven, and OODDP is just one example of where game developers are allowing data to drive application behaviour. One advantage of a data driven approach is that many independent titles can be described via changes to game content data, with very little low level programming required. Data driven methods have also been highly successful in the recent evolution of rendering hardware and software technologies, where a number of high level notations have been developed for the definition and subsequent control of a diverse range of visual effects and rendering techniques, as described in the following section.

2.3.2 Data Driven Rendering Pipelines

In the context of computer games, *rendering* is the process of forming a two-dimensional (image) representation from the description of the game environment. In most cases, the rendered image is presented to the user via display hardware such as a television or monitor. The vast majority of game environments are two or three dimensional, and range from the representation of traditional game boards (such as chess), through medium-sized situations containing multiple kilometres, to vast spaces consisting of solar systems, galaxies and even universes. The game environments themselves will typically manage spatial positions, extents and relationships for all objects in the game, plus their runtime states and logic in accordance with the game rules.

Almost all games are interactive, maintaining an internal representation of the game environment which may be modified in response to a variety of user inputs and logical rules. As the game environment changes, a new visualisation is presented to the user. A typical frequency for these visual updates will lie between 25 and 60 frames per second in order to maintain the illusion of continuous animation between the individual frames. Interactive games are thus similar to the model-view-controller (MVC) pattern from software engineering [97], in that the model consists of the game environment, user input and logical rules constitute the controllers, and a rendering process provides the view.

The typical render pipeline, as employed by the majority of modern graphics hardware, can be seen in Figure 2.15. Today's consumer-level graphics hardware for desktop systems include dedicated memory for geometry and texture data, as well as powerful graphics processing units (GPUs) for the parallel execution of rendering pipeline calculations. While the rendering hardware is responsible for performing much of the computational work involved in producing visualisations, a number of layers exist above the hardware in order to facilitate the communication of client data and processing control. The various layers involved in the communication between client application and rendering hardware are shown in Figure 2.16. Hardware vendors provide driver software for their products, which are in turn built upon by graphics library developers. Graphics libraries provide a variety of abstractions and functionality that client applications may use in order to define visualisations. A

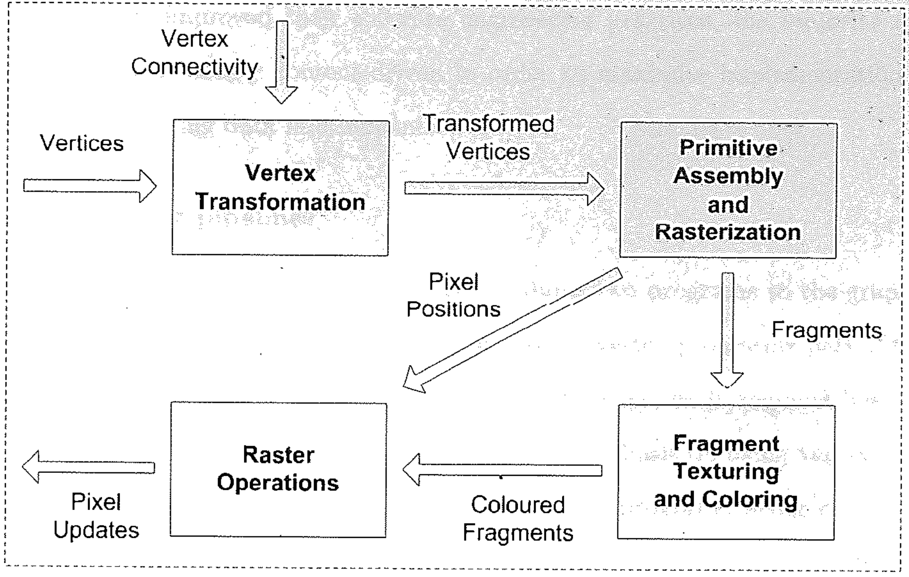


Figure 2.15: A typical hardware render pipeline.

typical graphics library will include a number of functions for the submission and subsequent control of geometry, textures, material properties, lights, cameras, and a range of special effects including fog and shadows. As graphics hardware capabilities have evolved, so has the range of functionality encapsulated by graphics libraries. The current leading graphics libraries are OpenGL and Microsoft's DirectX.

While visual representations are important for many computer games, certain genres are placing increasing emphasis on visual realism and detail. Developers are making considerable investments in order to incorporate complex surface representations, environmental features such as realistic lighting, shading and shadows, and both pre- and post- processing effects such as motion blur and depth of field. Many of the methods employed in such visualisations derive from academic study, and must be optimised or approximated in order for them to operate at interactive frame rates on current hardware. Meanwhile, hardware vendors are constantly improving the processing capabilities of their products, and exposing more diverse functionality to developers.

The continuing evolution of hardware capabilities, both low and high level APIs, and consumer demand, has resulted in a number of distinct advances in rendering technologies and methodologies.

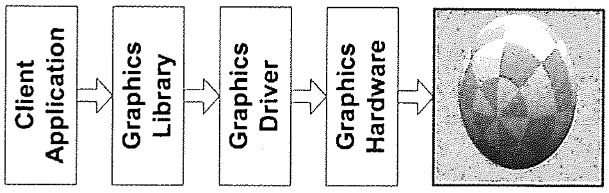


Figure 2.16: A layered illustration of the rendering pipeline, from the client application to the finished, rendered image.

As game developers have improved their software engineering practices, the majority of rendering pipelines have become increasingly content-driven in order to maximise implementation re-use and reduce the impact of escalating data requirements.

Programmable rendering pipelines

Modern rendering pipelines allow client applications to submit two programs to the graphics library, both written in dedicated assembly languages: one replaced the vertex processing part of the pipeline, while another was responsible for the rasterisation of *fragments*, where each fragment has the potential to be a pixel in the resulting rendered image. The continued emphasis on using vertex and fragment programs to determine the appearance of objects lead to such programs being collectively known as *shaders*. Although vertex and fragment shaders were initially severely limited in terms of size and functionality, further advancements in GPU technology later permitted longer, more advanced programs to be written, and higher level notations similar to imperative languages such as C and C++ to be used. The more successful high-level languages for programmable GPUs include High Level Shading Language (HLSL) for DirectX, GL Shading Language (GLSL) for OpenGL, C for Graphics (Cg) for both DirectX and OpenGL was developed alongside HLSL by nVidia, and Sh (originally SMASH) [98], which offers an alternative approach to manipulating GPU behaviour using high level code embedded in the application code. Figure 2.17 provides an illustration of the current technologies supporting the rendering pipeline.

While the power and flexibility of programmable rendering pipelines permit developers to create complex surface, pre-process and post-process effects, the logic required to fully implement these effects extends beyond the hardware and graphics library, and into the client application itself. The application must interact with the graphics library, via its API, in order to submit appropriate geometry, textures, lights and so on, as well as dedicated programs corresponding to material appearance and image processing effects. The selection of such programs, parameterisation of those currently selected, and their subsequent execution, are application dependent problems that cannot be solved by generic graphics library implementations.

Higher level concepts were introduced by graphics library developers in order to begin to tackle the problem of linking a range of client applications with the underlying graphics API and programmable rendering hardware. DirectX included support for the FX file format, which packages an individual material appearance or alternatively a single pre- or post- rendering effect. FX files include one or more *techniques*, which are intended to enumerate the various rendering methods available to the client. For example, one technique may only be compatible with the latest rendering hardware, while another may have differing data requirements. Each technique consists of one or more *passes* consisting of

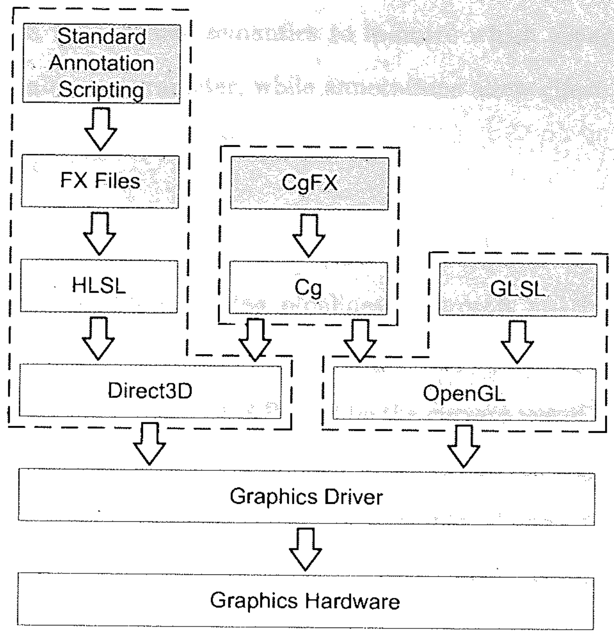


Figure 2.17: An overview of current software technologies supporting the rendering pipeline. Dashed areas denote software rendering abstractions and encapsulations provided by Microsoft's DirectX (left), nVidia's Cg (middle) and OpenGL (right).

vertex or fragment programs, which incrementally realise the intended material appearance or effect.

Client applications communicate with FX files via an FX API, which defines a layer of abstraction above the traditional data submission and control provided by older graphics libraries. Existing FX APIs include the ninth version of DirectX, while NVIDIA's CgFX framework [99] provides similar functionality for applications using either DirectX or OpenGL. An FX API will typically support the submission of data, selection and execution of techniques and passes, plus a range of functions for accessing the FX file's *semantics* and *annotations*.

Semantics are labels for FX variables, structures and parameters that indicate their intended use to the client application. To illustrate, the range of semantics include *POSITION* to indicate that a given variable represents a position in N dimensional space, while *NORMAL* and *COLOR0* correspond to surface normals and color values respectively.

Annotations are per-variable structures containing further variable assignments. For example, a floating point type `brightness` variable may specify an annotation including the statements `float min = 0.0f;` and `float max = 1.0f;`. Like semantics, annotations are accessible via the FX API, although unlike semantics they are commonly used to communicate details to an application that do not directly relate to the rendering process. For example, an application may use annotations to drive the placement, appearance and use of user interface elements controlling certain FX variables.

Together, FX semantics and annotations provide a standardised method for connecting client applications to FX files. FX files include semantics to indicate which aspects of client data should be assigned to each FX variable or parameter, while annotations allow effect authors to drive certain aspects of FX aware application functionality.

Data-driven rendering pipelines

Building upon today's programmable rendering pipelines, Microsoft are in the process of releasing information regarding their Standard Annotation Scripting (SAS) standard, which is expected to form part of the tenth major revision of the DirectX API. While the current use of FX files allows developers to supply data to the rendering pipeline in a very uniform and data-driven way, combinations of FX files must still be managed by application specific code. For example, shadow mapping [82] is a popular technique for generating shadows in rendered scenes. The shadow mapping technique requires all geometry to be rendered twice: once from the viewpoint of the shadow casting light source, and once from the viewpoint of the camera. On the second pass, a function checks if each fragment is in shadow or not, and renders appropriate pixel color values. In order to perform shadow mapping without the use of SAS scripts, the client application must prepare each rendering pass explicitly, and must also stipulate that the shadow checking function is to be called during the second pass.

SAS defines a strict standard for the use of semantics in FX files, as well as a powerful scripting language that is embedded in FX annotations. Applications conforming to the SAS standard must comply with its use of semantics, binding data of the correct type, range and format to the FX variables and parameters with corresponding SAS semantics. Furthermore, SAS compliant applications must also provide runtime behaviour for its range of script commands, and it is suspected that the next release of the DirectX API will both enforce data binding and supply appropriate SAS scripting functionality.

SAS scripts are embedded in FX string annotations, and may be attached to the file's various techniques and passes. For any given FX file, a global variable with a `STANDARDSGLOBAL` semantic holds the script's entry point as a string annotation. SAS scripts are responsible for performing a number of tasks, including selecting and clearing the current render target, assigning values to FX file global variables, performing a given technique or pass, and calling another FX file's SAS script.

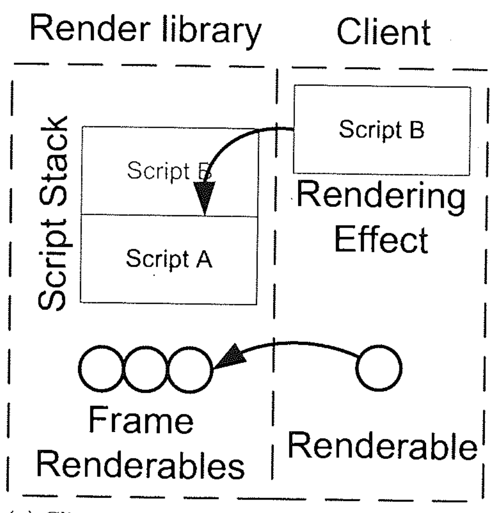
More complex rendering effects, such as the shadow mapping technique described above, are facilitated by the combination of FX scripts. To this end, SAS assumes the existence of a data structure with first-in, last-out (stack-like) behaviour capable of storing multiple scripts. While the script stack may be managed by an appropriate rendering library, the client application is responsible for pushing and popping FX scripts to and from the stack, as shown in Figure 2.18a. In order to render the scene, the entry point of the topmost script in the stack is located and executed; see Figure 2.18b. SAS script statements are executed in sequence until a `callexternal` command is encountered, at which point control flow passes to the entry point of the next script down in the stack, as illustrated by Figure 2.18c. If the script at the bottom of the stack includes a `callexternal` statement, then the rendering library is required to render scene geometry, which may in turn result in the execution of additional FX files and SAS scripts as geometry appearance is specified. This is shown in Figure 2.18d. When the end of an FX script is reached, control flow returns to the next script up in the stack. If the end of the topmost FX script is reached then control will return to the rendering library and client application as shown in Figure 2.18e.

The use of a stack for FX scripts provides a simple abstraction for calls to other FX scripts, and thus allows complex effects involving multiple scripts to be defined, without the need for complex coordination from the client application. The FX stack also allows pre- and post- processing effects to be defined in a very flexible way: if a given FX scripts performs its own processing before including a `callexternal` statement, then it defines a pre-processing effect and will be performed before any geometry is processed; the script calls `callexternal` before performing its own processing, then it represents a post-processing effect and will take effect after all geometry has been rendered.

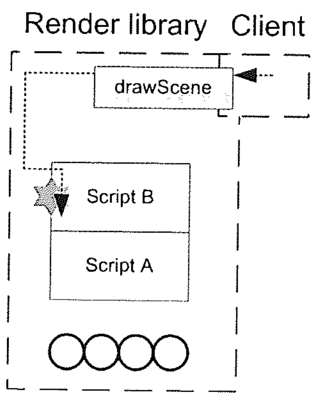
The use of rendering technologies to produce animated visual representations of spatiotemporal descriptions has become prolific in a wide range of computer game and academic applications. Early requirements for the efficient throughput of increasing numbers of geometric primitives led to the development of dedicated rendering libraries supporting higher level abstractions. More recently, growing consumer demand for more complex, detailed, dynamic and realistic scenes has resulted in modern rendering pipelines becoming more flexible to custom processing. In contrast to the fixed functionality pipelines of the past, today's data driven rendering pipelines allow a variety of low level hardware operations to be driven using high level notations forming part of application data. Despite a persistent need for processing speed and efficiency, modern rendering pipelines clearly illustrate the flexibility and expressiveness afforded by incorporating data driven techniques.

2.3.3 Summary of Computer Games Technology

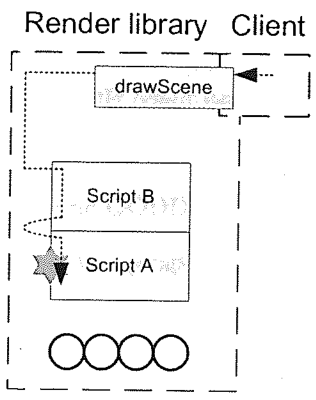
OODDP has become a popular methodology for game developers, who replace static inheritance relationships in software structures with ownership relationships that can be dynamically driven by data at runtime. However, the use of OODDP remains confined to application *content*, where it is limited to describing the contents of the gaming environment, including both static and dynamic elements. OODDP has not been applied to the definition of software structures within the game engines themselves, which instead act as frameworks taking supporting roles in data-driven applications. The most likely reasons for this limitation are the additional memory and processing requirements of OODDP



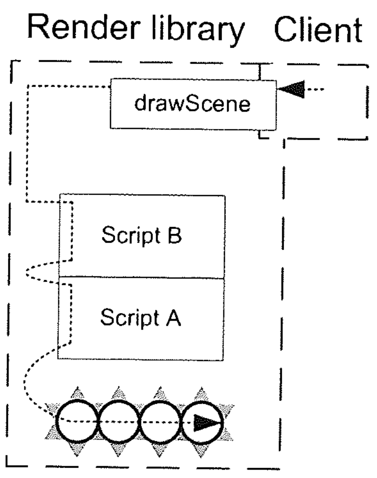
(a) Clients are responsible for providing renderables, pushing scripts to and popping scripts from the rendering library's effect stack.



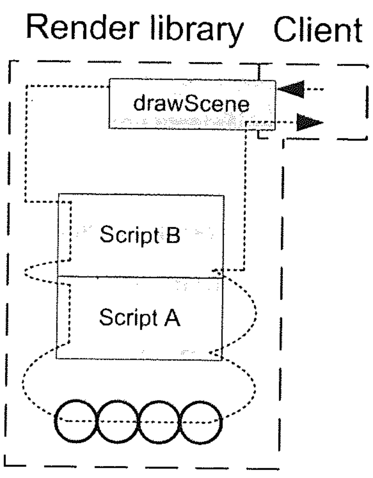
(b) When drawing the scene, control flow passes to the topmost effect in the stack.



(c) A `callexternal` call in script B results in control flow passing to the next script down in the stack.



(d) A `callexternal` call in script A results in this frame's renderables being drawn.



(e) As scripts A and B are completed, control flow passes up through the script stack and ultimately back to the client.

Figure 2.18: An illustration of the use of an SAS script stack to support complex rendering effects. Dotted lines and arrows denote the runtime flow of execution.

implementations, which would prohibit its general use to develop processor intensive interactive applications like modern game applications. In the context of more mainstream commercial and academic software development, there may be opportunities to benefit from the flexibility and expressiveness of OODDP software where such memory and processing requirements are less prohibitive.

Furthermore, OODDP remains an *applied* methodology, with no formal theory and no explicit connections to related fields. OODDP has yet to be investigated outside of the game development context, although similar technologies exist, as described below.

Code generation performs a similar process of transforming higher level descriptions into lower level software representations. In the context of the MDSD approach, OODDP could be considered a transform responsible for mapping a platform independent description of an OO type system to a platform dependent model providing component or object centric OODDP implementations. The component and object centric models would provide further mappings to lower level database and component based implementations, respectively. This relationship between OODDP and MDSD suggests that there may be potential benefit from combining the two technologies.

Despite their superficial similarity, there is a clear difference between the code generation process and OODDP approaches. The ultimate output of code generation is low level software code, typically producing class interfaces and definitions to be statically compiled as the result of transforming higher level descriptions into lower level implementations. By contrast, OODDP is a runtime factory process that instantiates and interconnects small scale data or object based components in order to form a higher level object oriented type system. Where code generation produces static software code, OODDP dynamically creates runtime topologies.

Interpreted languages offer very similar functionality to OODDP, particularly those supporting OO or pseudo-OO type systems. Many interpreted languages support the dynamic configuration of OO types and instances from high level descriptions.

However, while interpreted scripts often form part of application data, there is a clear distinction between the roles of interpreted scripts and OODDP configurations. For example, interpreted languages provide lower level functionality than that delivered by OODDP, concentrating on much smaller state and behavioural elements including individual variables, function definitions and code statements. By contrast, OODDP focuses on the deployment of black box components; there is no direct manipulation of state, and runtime behaviour is influenced via high level component configuration and by interconnecting component interfaces.

While OODDP operates on more abstract and discrete components than those *typically* manipulated by interpreted languages, the latter technology is capable of supporting OODDP's higher level concepts through the use of appropriate extensions and APIs; indeed, interpreted languages such as Python and Lua would offer an ideal platform for OODDP implementations. There remains, therefore, some blurring of the distinction between the two approaches.

Composition languages are perhaps the most similar mainstream software engineering technology to OODDP's fundamental ideas: both approaches focus on component instantiation and interconnection in order to define coherent software structures. However, there is a clear difference in the application of the two technologies.

The main use of OODDP is to describe an OO type system, where each type consists of a number of configurable elements. The application of OODDP is typically limited to application content data, and focuses on the definition and configuration of OODDP instances that will subsequently appear as visible (and often interactive) elements in the runtime application.

Conversely, composition languages focus on the explicit deployment and interconnection of component instances. While a component configuration may define a major portion of application functionality, the particular configuration is often invisible to the user. Furthermore, composition languages are explicitly developed and applied for the definition of component deployment and composition, whereas the implementation of an OODDP type system is not limited to a component based approach.

A further distinction is introduced by the level of abstraction and granularity adopted by OODDP and composition languages. Component based approaches in mainstream industry and academia typically make use of components encapsulating concepts at the level of one or more objects. While a small number of composition languages support smaller concepts such as functions and procedures, the vast majority of components provide a variety of abstractions ranging from objects to entire libraries. A composition language supporting the OO paradigm will often define a thin wrapper for such components by directly mapping the component's methods to a similar interface definition in the composition language for further manipulation.

By contrast, OODDP builds an abstract OO type system that incorporates components encapsulating small scale concepts such as single operations or aspects of a class. The capabilities and behaviour of an OODDP type is the union of the functionality provided by its constituent components.

As with OODDP, it is clear to see that the evolution of modern rendering pipelines is moving toward the use of data-driven methodologies. Early efforts to provide a powerful pipeline connecting application content to low-level hardware functionality relied upon the client application to perform many tasks that were later encapsulated by graphics libraries. More recently, graphics libraries and FX APIs have incorporated high-level languages that can be used to control hardware behaviour, while guiding the use of application data through the use of semantics and annotations. Today's rendering technologies allow application data to drive the behaviour of the rendering pipeline.

The movement toward data-driven methodologies will continue with the introduction of Microsoft's standardised annotations and semantics, which are embedded into the definition of the effect and appearance descriptions that form a significant part of today's game application content. The expressiveness and functionality of FX files is rapidly approaching the capabilities of traditional interpreted languages; their operation relies upon a fixed set of compliant data bindings and runtime interpreted behaviour as supplied by client applications and future graphics libraries. Meanwhile, the metaphorical rendering pipeline supported by modern rendering APIs is becoming increasingly concrete, with well-defined data bindings for client applications, and standardised runtime functionality provided by SAS compliance and programmable rendering hardware.

2.4 Context Summary

The role of data in applications is changing. As discussed in Section 2.1, application data is becoming increasingly empowered to incorporate type information, data manipulations, contextual documentation, meta data, and range of other aspects that have traditionally been recognised as the responsibility of application software. Data is also beginning to determine or otherwise significantly influence application behaviour, a growing trend that is embodied by the DDDAS paradigm, which incorporates elements of application data to enhance its runtime functionality. Although both GIS and DDDAS incorporate a particular focus on modelling the world around us, neither have yet to fully realise the benefits of data driven behaviours: the vast majority of GIS operations for data creation, visualisation and manipulation remain fixed over the lifetime of the GIS software. However, a small number of GIS projects have begun to include data driven aspects, and it is becoming increasingly likely that the GIS community will embrace a more data driven approach in the future [100].

Data driven paradigms advocate the use of expressive, human readable notations to drive software behaviour at high levels of abstraction, which reflects a similar movement in mainstream software engineering. The component based software development methodology is becoming increasingly popular and widespread, as software developers benefit from the increased implementation reuse, flexibility and extensibility that the object oriented paradigm failed to provide. While independently developed and deployed components provide runtime computational behaviour, high level composition and configuration languages drive their topology to form connections, communications, and collaborative functionality. Although such configurations are typically employed to drive component compositions in a bottom up fashion, a small number of methods support runtime reconfiguration at the structural or architectural level [101, 102, 103].

In contrast to the GIS field, the interactive spatiotemporal scenes of modern computer games commonly incorporate a number of data driven technologies. As part of a movement toward more effective software development methodologies, game developers have introduced a range of related data driven techniques in order to keep abreast of changing demands from both game designers and consumers. For example, data describing application content may include an expressive object oriented notation to drive software structure in a similar way to that of composition languages. Such notations have proven to be particularly popular, as their expressive high level nature means they are commonly open to manipulation via tools and non-programmers, including game designers and members of the consumer community. Meanwhile, rendering hardware vendors and graphics library developers provide powerful abstractions allowing rendering pipeline behaviour to be driven by high level languages embedded in application content.

While a future movement toward data driven approaches in GIS is likely, a significant hurdle to overcome is the fixed functionality defined by a chosen data model. The introduction of DDDAS technologies would add a dynamic aspect to data manipulations, but the degree of change in DDDAS has limitations: dynamic behaviour is typically implemented by modifying the parameterisations of operations in response to runtime data. There is clear potential benefit from taking a step beyond the data driven dynamics supported by DDDAS, to incorporate manipulations of the application software itself. This could be achieved by incorporating component based approaches, defining operations over a meta model of the software that in turn drive the (re)composition of component topologies.

However, additional benefit may be gained by merging mainstream component based methodology with the object oriented data driven notations used by game developers. The resulting notations would be able to drive runtime software composition at runtime via an empowered notation built into application data. Furthermore, a combined approach would allow the same human readable notation to be used to define both application content and application software. Finally, the notation would be expressive enough to be open to manipulation via tools and non-programmers, potentially bringing the popularity and ease of consumer based computer game customisations to the GIS community.

Chapter 3

The Fluid Component Framework

The Fluid framework [104] combines OODDP technologies from game development with mainstream CBSD methodologies in order to form a component framework and composition language with a particular focus on configuration, extensibility and flexibility. The Fluid framework aims to emphasise implementation reuse by incorporating both a component based approach, which aims to achieve reuse through the development of software from prefabricated binary components, and OO style composition inheritance, as supported by the OODDP techniques described in Section 2.3.1. By bringing these elements together, the Fluid framework supports an expressive, object-oriented notation for the description of component applications.

Section 3.1 provides an overview of the motivation, aims and objectives for the design and development of the Fluid framework, while Section 3.2 presents the three tiers comprising the Fluid framework, including a high level description of each tier as well as relevant lower level aspects of their design. Meanwhile, Section 3.2.4 describes the Fluid executable, which forms the entry point for Fluid applications. Section 3.3 presents a complete example of a Fluid application, bringing together the individual descriptions for Fluid's three tiers and executable. Finally, Section 3.4 summarises the description of the Fluid framework.

3.1 Motivation, Aims and Objectives

The development of the Fluid framework was primarily motivated by the limitations of the project's earlier research, which attempted to introduce a data driven approach to geovisualisation. Bringing together a number of concepts from Chapter 2, the Fluid framework's prototype application aimed to deliver a flexible and extensible software framework capable of producing real-time, near photorealistic visualisations for geospatial scenes. The focus of this earlier work was to combine data driven programming and rendering techniques to form a data driven software framework providing geovisualisation functionality.

As described in Section 2.1.1, the field of geovisualisation is beginning to incorporate data driven elements as part of a small number of academic projects. However, commercial geovisualisation solutions have yet to adopt data driven methodologies. Indeed, is has been observed that many GIS have limited visualisation capabilities, and often utilise separate technologies to provide rendering functionality [14]. Meanwhile, the introduction of more detailed and realistic renderings [105] and the visualisation of dynamic phenomena [106] remain issues for ongoing research. Alternatively, a number of game engines focussing on the representation of geospatial scenes have been proposed as possible geovisualisation platforms [86, 87]. The idea of leveraging certain game engines in order to support geovisualisation functionality seems attractive: as described in Section 2.3, today's computer games technology is capable of producing incredibly immersive, realistic and interactive environments incorporating geospatial data at various scales. However, while it has been demonstrated that existing game engines can provide visualisations for a wide range of applications [88], geovisualisation and game-based visualisations have differing underlying motivations: while modern game titles focus on player immersion, atmosphere and interaction, geovisualisation is used to communicate geographically referenced information.

The Fluid prototype aimed to bridge the gap between computer games technology and traditional geovisualisation solutions by incorporating a simplified game engine targeting general geospatial scenes. The prototype's reduced game engine included only the most relevant functionality related to simulation and visualisation; superfluous features such as artificial intelligence and storytelling were not considered. By building upon existing game development techniques, the prototype aimed to provide additional visual quality, detail and realism as seen in modern game titles. While a description of the Fluid prototype's objectives and high level design can be found elsewhere [107, 108], the following paragraphs highlight the most relevant achievements of this research:

Data driven scene system The prototype's simulation content was defined using a hybrid OODDP system as described in Section 2.3.1; the scene system's use of OODDP was influenced by a range of game object system designs [92, 93, 109, 110, 96]. The Fluid prototype's scene system applied OODDP outside of the game technology field in order to allow application developers to describe the functionality and runtime behaviour of scene content using configuration data. However, the prototype's application of OODDP remained similar to that seen in the motivating literature.

Data driven rendering system The Fluid prototype also included a data driven rendering system, as described in Section 2.3.2, supporting similar functionality to an anticipated Direct3D release. However, while Microsoft's Direct3D API offers a complete implementation of a data driven pipeline in a single proprietary package, the Fluid prototype delivered equivalent functionality using open alternatives: OpenGL provides low-level rendering capabilities, while nVidia's Cg and CgFx languages present abstractions for individual shader programs and effect representations respectively. The Fluid prototype introduced an additional layer supporting a subset of SAS commands, which allowed effects to be connected together in order to form more complex effects for geometry materials and pre- and post- processing. The Fluid prototype's render system implementation is the first to deliver a complete open data driven rendering pipeline from high-level SAS abstractions to low-level rendering functions. Furthermore, Fluid's implementation is platform independent, while that provided by Direct3D is restricted to the Windows platform. Due to its potential benefit to the visualisation community, the functionality defined by the Fluid prototype has been developed further in order to create the PirateHat rendering library [3]: a C++ static library that can be used independently of the Fluid project.

The Fluid prototype inspired a second software development iteration with a more concentrated focus on mainstream computer science methodologies. The Fluid prototype's implementation had shown that producing interactive, near photorealistic visualisations for geovisualisation was unlikely due to certain limitations of the prototype's design and the poor data availability common to GIS applications. Consequently, the research focus was modified to have a more narrow scope, and to instead focus on utilising computer game technologies in order to introduce a flexible, extensible and dynamic software architecture to the field of GIS.

In order to achieve this aim, the Fluid framework would build upon the Fluid prototype's OODDP implementation by incorporating the flexibility of component based software development. A component based approach would increase both flexibility and extensibility by allowing each Fluid application to introduce its own domain and semantics as part of the selection, composition and configuration of Fluid components. A component based Fluid framework would be responsible for supporting the development of Fluid applications by both aiding and enforcing the use of valid Fluid components, as well as their correct composition and the configuration of their data driven behaviours.

Contemporary component based approaches make use of high level notations to drive the selection and composition of components in order to define software applications. A number of these composition languages have a similar appearance to the OODDP technologies available to game developers. In order to understand the relationship between composition languages and OODDP approaches, a review of the CBSD field concentrated on the following issues:

To what extent do existing composition languages make use of OO concepts, abstractions and manipulations? A wide range of composition languages regularly make use of OO types, the instantiation of OO types, OO interfaces and encapsulation. These concepts commonly enhance the functionality and expressiveness of the concepts embodied by the composition languages. A number of technologies such as delegates [111] and mixins [112] also exist to support the use of OO encapsulation and inheritance across individual components. Furthermore, a number of ADLs directly support OO types, instances, interfaces and inheritance, albeit at a higher level of abstraction.

However, support for these concepts in the composition languages themselves is often limited, with many composition languages acting as thin wrappers around equivalent concepts and operations provided by underlying OO components. While such languages allow OO inheritance to be established between components, these relationships are commonly established between classes contained within components, and not between the components as they are represented in the composition languages themselves.

To what extent do existing composition languages focus on configuration, parameterisation and data driven approaches? The vast majority of composition languages define a high level textual notation for defining software applications as the selection, interconnection and parameterisation of components providing computational encapsulations. However, none of the composition languages encountered have the same emphasis on parameterisation and data driven behaviours as seen in DDP and OODDP. Instead, many composition languages focus on incorporating additional behavioural capabilities, such as the inline invocations seen in CoML [33], or formal notations, such as those seen in PICCOLA [26].

Component customisation remains an open topic of research in CBSD. Proponents such as Schultz [113] advocate the specialisation of black box components, but at the implementation level, which could prove obtrusive to the overall CBSD methodology of independent component development. A similar approach is taken by Weis [114], who advocates the application of component customisation *before* they are distributed. A wide range of studies have investigated the use of component wrappers to provide customisation during component deployment [115, 116, 117], although there has also been a plea for grey box components that can be more easily customised [29].

A small subset of research looks at the use of smaller computational encapsulations. For example, Mirza [118] demonstrates that small grained components may aid rapid application development, while Lorenz [119] states that smaller components at the granularity of objects have a closer correspondence with artefacts of OO design and implementation, and may be more approachable when developing component based systems. Meanwhile, Hamlet [120] discusses component scale, concluding with the argument that smaller components can be useful when precise understanding of their operational semantics is important, for example in the case of unit testing. Reussner [121] discusses the tradeoffs of both large and small components, and concludes that component *contracts* should be adaptable in order to increase the reuse of components at various scales of granularity.

The results of this initial research into the field of CBSD introduce a number of interesting research questions, including:

What is the effectiveness of a novel composition language combining OODDP and mainstream CBSD approaches? What benefits and drawbacks does OODDP introduce? How does the type system and OO inheritance introduced by OODDP relate to existing approaches such as delegation and mixins?

Can a novel composition language incorporating OODDP be used to define overall software structure? Alternatively, should the application of OODDP be limited to software content, as already demonstrated by game engines?

Relevant reference materials [122] and dedicated workshops[1] provide an overview of the CBSD field, and also define a number of accepted concepts and terms. In particular, those materials giving a high level appreciation of the past, present and future themes in CBSD [43] highlight the limitations, drawbacks, and unexplored aspects of current research. However, the design and implementation of the Fluid approach has been influenced by a more narrow range of related work. Described below is a small selection of the reviewed literature. Although a wide and varied range of work has contributed to the design of Fluid's component framework, only the most influential or distinctive publications are highlighted here.

Jiazzi [123] provides an example technology for writing component based software using Java packages. Interestingly, Jiazzi does not make use of a higher level composition language; instead, Jiazzi relies upon the deployment and reflection mechanisms of the Java language, plus additional support for external linking and separate compilation, to define components that may be hierarchically assembled and executed by a Java Virtual Machine. The range of concepts supported by Jiazzi present an interesting insight into how implementation languages, additional compilation steps, code generation, and higher level manipulations can be combined to provide a component based solution. A related publication by the same authors [124] also presents a range of essential technical properties for component systems.

[1]For example, see `http://www.ict.swin.edu.au/personal/mlumpe/` for an index of past events related to the Workshop on Composition Languages, and `http://research.microsoft.com/~cszypers/events/WCOP2007/` for the 2007 meeting of International Workshop on Component-Oriented Programming

PICCOLA [26] incorporates the explicit separation of component implementation and composition in its support of the paradigm `Applications = Components + Scripts`. XCompose [41] further refines this as `Applications = Components + Composition Language`, and employs various XML technologies in order to form its underlying composition model. The clear need for separate abstractions for component composition greatly influenced the design of Fluid's composition language and underlying framework.

While Jiazzi and PICCOLA provide an influential introduction to some of the higher level concepts of component based approaches, a variety of lower level concerns and possible solutions are also presented, highlighting a range of potentially important limitations to be considered by Fluid's design and implementation. For example, the enforcement of component encapsulation behind restrictive interfaces, as well as carefully designing interface invocation semantics, could improve the quality and reliability of both components and their compositions [125]. Also, it may be important to consider the distinction between control and data flow in component compositions [126].

In addition to the more theoretical studies discussed thus far, a number of publications also describe the *application* of component methodologies in a wide variety fields, including those related to the visualisation of 3D scenes. Such publications give an important insight into the overall methodology of developing component based software, as well as the motivation for integrating a component based approach into a range of problem domains. For example, Contigra [39] is a component based architecture for the development of interactive 3D applications, which may be standalone or web-based. Of particular relevance to Fluid's focus on geovisualisation is Contigra's use of a scene graph to describe its 3D scenes, including the participation of JavaBean components. The syntax and semantics of Contigra's various XML documents also motivated the design of similar documents in the Fluid framework.

The two most influential technologies for the design of the Fluid framework are VHD++ and CoML. VHD++ [42] is a component framework supporting interactive, real-time 3D simulations making use of a combination of game, virtual reality and augmented reality technologies. There are obvious parallels between the domain of VHD++ and Fluid's geovisualisation focus. Furthermore, both frameworks face the same opposing issues of performance and generality, and aim to resolve this disparity via the application of game derived technologies. Consequently, VHD++ has had a profound impact on the design of Fluid's underlying component framework. For example, both VHD++ and Fluid make a distinction between data and behavioural encapsulations, as well as supporting procedural and event-based inter-component connections. Fluid also makes similar augmentations to C++ in order to support additional type information, reflection, and improved heap memory management. Furthermore, the XML based composition language employed by VHD++ was also considered when designing Fluid's own composition language.

However, the composition language described in Section 3.2.3 bears a stronger resemblance to CoML [33], a platform independent XML based composition language forming part of a larger framework for component composition [36, 127]. Fluid's composition language is heavily influenced by CoML for a number of reasons. The most important reason for this is that CoML is an *operational* composition language, forming part of a larger study into component distribution and composition, and based on requirements stipulated by the earlier work of Nierstrasz and Meijler [30]. In addition, CoML has particularly clear syntax and semantics, and a focus on connection based programming with black box components. Finally, CoML also incorporates an appropriate amount of type information, without becoming overly verbose, in order to support components written in a variety of implementation languages. These features made CoML an ideal foundation for Fluid's customisations.

The design for Fluid's composition language was initially built upon a simplified version of CoML, most notably discarding those elements describing runtime behaviour, but retaining those elements pertaining to composition selection, connection and configuration. Certain aspects of VHD++, as well as minor additions from other work discussed here, have also been incorporated.

The Fluid component framework represents a second iteration of the design and implementation process, which builds upon the Fluid prototype by incorporating a CBSD approach. The limitations of the Fluid prototype motivated an improved design that incorporated aspects of a wide range of computer science and game technologies. For example, implementation and integration languages such as C♯, Java and Lua informed the redevelopment of Fluid's lower level functionality. Meanwhile, the design of Fluid's higher level abstractions incorporate a significant change in the project's overall research aims. In particular, Fluid's composition language combines component based technologies with data driven techniques in order to contribute a novel CBSD approach to mainstream computer science.

The design of the Fluid framework is described in the following section.

3.2 Design

An illustration of the Fluid component framework's architecture is shown in Figure 3.1; its three tiers are described below: Section 3.2.1 presents an overview of Fluid's lower tier systems and their respective responsibilities, Section 3.2.2 defines Fluid's component model, and Section 3.2.3 describes Fluid's composition language. Section 3.4 gives a summary of the material in this chapter.

Figure 3.2 provides an alternative view of the Fluid framework, showing the runtime relationships between its tiers and systems. An XML document describing a particular Fluid application is parsed

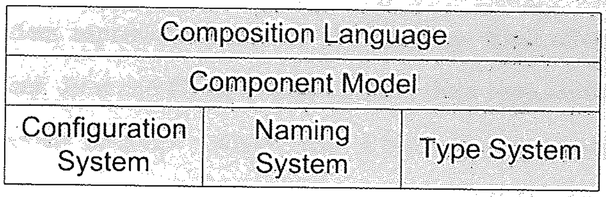


Figure 3.1: An overview of the Fluid component framework's architecture. Three fundamental systems form the lower tier (see Section 3.2.1). These are built upon by a component model (see Section 3.2.2), which facilitates the definition of a high level composition language (see Section 3.2.3).

by Fluid's composition language tier. The composition language tier drives further operations in the component model and configuration system. As the document's XML elements are processed, a range of primitive and OODDP types are introduced to the framework. Runtime objects corresponding to these types are composed in order to form a complete Fluid application, which is finally executed by the Fluid framework.

3.2.1 Type, Naming, and Configuration Systems

Fluid's Type, Naming and Configuration systems define a low level C++ platform to support the higher level concepts introduced by the component model and composition language. Fluid's lower tier is responsible for augmenting the capabilities of C++ so that the framework's higher tiers may be defined. The Type system provides a range of standard data types, additional runtime type information to that provided by C++, limited support for type reflection, as well as a range of new type checking operations. Meanwhile, the Naming system supports a unified deployment space for a range of data and behavioural representations that collectively define Fluid applications. Finally, the Configuration system gives limited support for converting XML descriptions containing type system concepts to their corresponding naming system representations.

The functionality provided by Fluid's lower tier is influenced by a number of other implementation languages offering capabilities that C++ lacks. For example, the primitive data types supported by

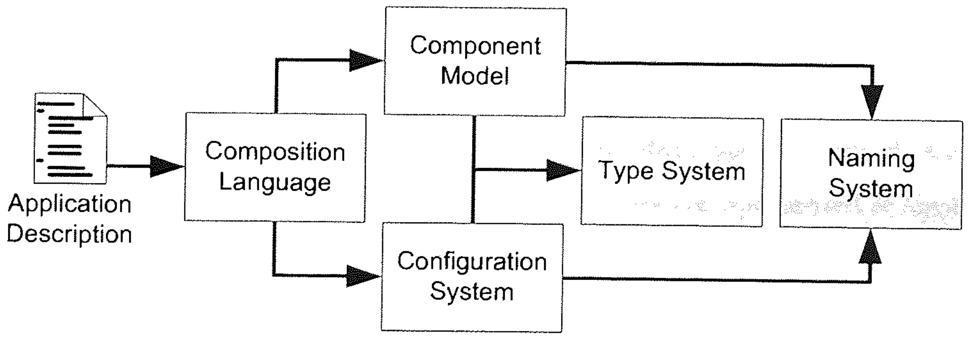


Figure 3.2: An illustration of the runtime relationships between Fluid's tiers and subsystems. Arrows depict the flow of information and control; for example, Fluid's composition language drives the behaviour of the component model by providing OODDP type and instance descriptions.

C++ have platform dependent representations, and the language itself offers limited run time type information; by contrast, both Java and C♯ have standardised data representations and very powerful type information and reflection facilities. Meanwhile, XML and Lua's tables have influenced the development of Fluid's naming system. Finally, C♯ and Lua support the use of data representations to describe software structure: C♯ supports conversions between C♯ class descriptions and XML, while Lua literature [128] describes a flexible way to store Lua tables in plain text format. It is clear that migrating the Fluid framework to an implementation language more suited to the development of component based software would make certain parts of the lower tier redundant. However, the Fluid framework's development was based on the prototype's implementation, and initially reused much of its functionality.

Type System

The type system is responsible for providing a uniform concept of runtime type information for all types that are *visible* to the Fluid framework. Visible types include values stored in the naming system, values communicated between component instances, the functions and events used for inter-component communication, values used to configure component instances, individual component instances being managed by the framework, and OODDP type and instance descriptions. Visible types thus facilitate interoperability for application configuration and runtime functionality. Those types forming part of component implementations or Fluid's system implementations are *invisible* to Fluid's type system, and are not necessarily supported by the type system.

Fluid's visible types consist of *primitive types* for simple values (variables), primitive *manipulations* for more complex data representations, and *signature types* for event and function signatures, as described below:

Primitive types Fluid primitives provide a range of fundamental concepts for the storage and runtime representation of values. Primitive types are derived from a subset of XML schema types in order to take advantage of an existing standard. The primitive types therefore include support for dates, times and durations; strings and certain other types that can be represented using strings; single and double precision floating point numbers; and both signed and unsigned integers of various sizes. Fluid adds a `nil` type, which represents uninitialized or empty values, and may also be used in situations where a type is unknown, superfluous or irrelevant[2].

The Fluid framework stipulates a range of C++ types that correspond to its supported XML schema types. The majority of these C++ types have been chosen in order to provide a standardised representation for numerical values. For example, the C++ Boost library provides a

[2]For example, in some contexts `nil` may be used in place of the C++ `void` keyword.

range of types that guarantee the minimal requirements of 8, 16, 32 and 64 bit integer representations. While the *exact* representations for these data types may vary, Fluid is able to support their corresponding XML schema types on a wide range of platforms by using such Boost abstractions. There are at present certain limitations: the current implementation of the Fluid framework does not support types with infinite range, such as the `decimal` and `integer` types; similarly, certain specialisations of the `string` type, including `token` and `NCName`, are not currently considered and will be converted to Unicode strings by the Fluid framework. However, note that such type conversions do not affect XML schema validation, which occurs prior to any changes made by the framework.

Manipulations Fluid supports more complex data descriptions using manipulations of the above primitive types: `Sequence` holds multiple values with a repeating type pattern[3], `Choice` stores a single value of a fixed range of types[4], and `Optional` will hold either a value or `nil` at any given time[5]. Manipulations may be used recursively, and may form complex type descriptions, such as a sequence consisting of an optional boolean followed by a choice of 32-bit integer or single precision floating point number.

Signature types use the primitive types and manipulations given above to describe the return values and parameters for event and function signatures. The type system is thus able to check inter-component connections for type safety by comparing the type information for connected signatures.

Fluid provides a `Value` class for defining and communicating primitive types, including their manipulations. The Value concept is heavily influenced by Lua variables, which are dynamically typed according to the value they are currently holding. In order to mimic this behaviour, the Value class makes use of a discriminated union concept[6] so that each instance may hold a variety of primitives and manipulations during its lifetime. The content of a Value may be accessed by first inspecting the Value's current type, and then calling a type casting template method with an appropriate type parameter:

```
bool internalValue = aValue.as<bool>();
```

However, a more flexible method is to use a combination of the visitor pattern and C++ function overloading to provide a visitor with specialised behaviour for each type. A generic function may be used to collect those types for which no special behaviour is required. Value visitation is illustrated

[3]For example, a sequence of four strings, or a string followed by a boolean may be described.
[4]For example, holding either a boolean or a string.
[5]An `Optional<T>` is thus equivalent to a `Choice<nil,T>`.
[6]Value's discriminated union is provided by the Boost variant library.

in Listing 3.1; further information is available via the Boost library documentation[7].

Meanwhile, component functions and events, along with their connection and invocation, are supported in the Fluid framework by `Function` and `Event` classes, respectively. Although Fluid's Function and Event classes exhibit differing runtime behaviour, in the context of the type system, the two classes are very similar, as both Function and Event are encapsulations of procedural signatures.

Function and Event are in fact supported by two classes: one for the representation of the Function or Event itself, and another for connecting or subscribing to a given Function or Event instance. The former storage classes provide an encapsulation that wraps Function and Event signatures behind a uniform interface. These storage classes are not template classes; signature information must be hidden in order to provide a simplified abstraction for runtime storage in the naming system (see below). The latter access classes are responsible for providing type safe access to a given storage class' encapsulated Function or Event. These access classes *are* template classes, parameterised with the desired Function or Event signature, and their connection or subscription will fail if the given signature does not match that of the storage class' type. The storage and access classes use a generalised union type [129] in order to provide the required uniform encapsulations and type safe access idioms.

As well as providing type rich representations for Fluid's fundamental storage and communication concepts, the type system is also responsible for maintaining type information for any OODDP types that are introduced to the framework at runtime. Each OODDP type consists of a type name, optional parent type, and type description, all of which are provided by Fluid's composition language. The type system is required to maintain a collection of known OODDP types, and must also provide access to their parent types (if applicable) and descriptions. OODDP types are described as part of Fluid's configuration system (see below).

In a similar way to the Value class, the type system encapsulates its various type information

[7]http://www.boost.org

Listing 3.1: Visiting a Value instance in order to obtain type specific behaviour.

```
class IsManipulationVisitor : public boost::static_visitor<bool>
{
        // return true for manipulation types
        const bool operator()(const Sequence&) const
        {       return true;
        }
        const bool operator()(const Choice&) const
        {       return true;
        }
        const bool operator()(const Optional&) const
        {       return true;
        }
        // for all other types, return false
        template<typename T> const bool operator()(const T&) const
        {       return false;
        }
};

// visit a Value instance to determine if it is a manipulation type
bool isManipulation = boost::apply_visitor(aValue, IsManipulationVisitor());
```

representations using a single `TypeInfo` class, which makes use of the same discriminated union idiom. As with Value, the actual type held by a TypeInfo object may be accessed via a casting method or appropriate visitor, leading in some cases to additional type information such as the types held by a given Sequence, or the return value and parameter types for a given Function. However, the TypeInfo class interface also provides common operations such as testing for type equality, relative ordering of TypeInfo objects, and obtaining a descriptive string for each type.

Through `Value`, `Function`, `Event`, and `TypeInfo`, the type system provides a reflective, type safe means for components to communicate with the Fluid framework and each other. Furthermore, component configurations, deployments and connections may be checked at runtime, to ensure that such compositions conform to Fluid's component model as well as their own specifications. Further detail regarding Fluid's component model can be found in Section 3.2.2.

Naming System

The Fluid framework's naming system provides a central *blackboard* [130] for the storage of named objects. The naming system consists of an acyclic graph of associative containers, where objects are associated with a given name, and may be subsequently identified and accessed using that name. The most basic associative container is the *Namespace*, which is allowed to store other Namespaces, `Values`, OODDP type descriptions, or OODDP instance objects. OODDP instances are also associative containers, and may store nested OODDP instances, `Values` and component instances. Component instances form the final layer in the naming system, and may contain `Values`, plus any `Functions` and `Events` that are exposed by the component. The range of allowed types at each level in the hierarchy are illustrated by Figure 3.3. Note that the naming system itself holds a single Namespace instance providing the root of the naming hierarchy.

Namespaces, OODDP Instances and components are supported by a generic MappedStorage base class, which encapsulates the concept of an associative container. Classes inheriting MappedStorage may provide custom functionality, such as access to type identification, but will generally defer control of the associative container to the MappedStorage class. MappedStorage is a generic class taking a list of stored types as a type parameter. The resulting MappedStorage class specialisation maps string identifiers to instances of a discriminated union over its stored types. Methods for querying and manipulating the contents of the associative container are provided.

Due to the use of MappedStorage and discriminated unions, a wide range of operations over naming system contents may be implemented using various forms of C++ template meta-programming, including the type specialised visitation technique discussed above. A combination of visitors and overloading function definitions provide the vast majority of naming system functionality.

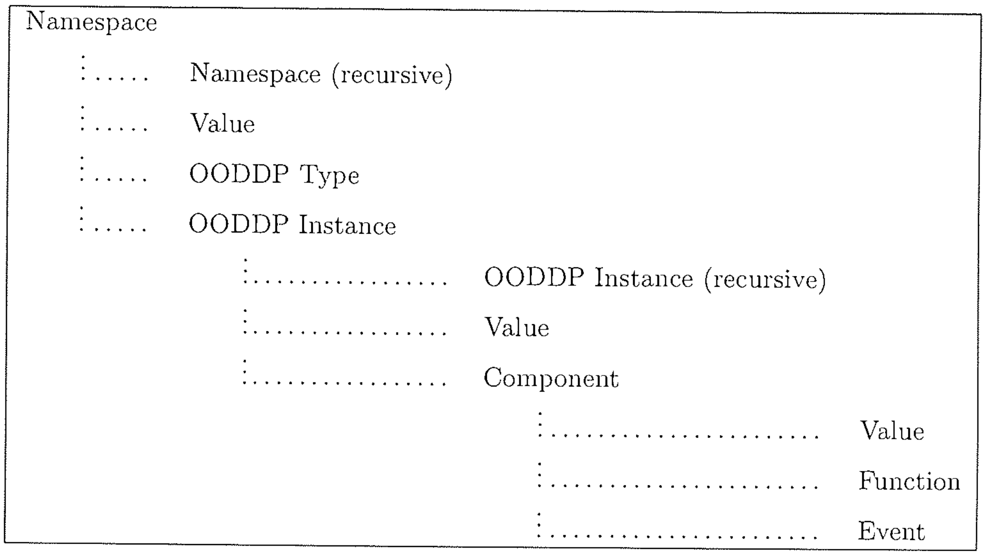


Figure 3.3: The concepts allowed at each level in the naming system hierarchy.

In contrast to more flexible deployment spaces, such as Lua tables, Fluid's naming system has a strict structuring to its stored concepts, as shown in Figure 3.3. For example, Fluid components may not be placed in namespaces, but must form part of OODDP instances. The hierarchy of types shown in Figure 3.3 is enforced in order to emphasise the use of object oriented design when developing Fluid applications: runtime behaviour must be encapsulated by OODDP type and instance descriptions. By forcing behavioural components into OODDP types and instances, Fluid's naming system forms the first part of a layer of object oriented concepts, which reside above those OO concepts supported by the C++ language. The restrictions embodied by the naming system's type hierarchy are built upon by Fluid's composition language to define an OODDP textual interface for the definition of component software (see Section 3.2.3).

Fluid's naming system supports flexible deployment by facilitating type-safe late binding. This is achieved via the use of *weak references* [131]. Weak references essentially allow objects to refer to one another via an intermediate proxy, before the objects themselves have been fully deployed: an object may create, during its configuration-driven initialization, a type-safe weak reference to another object's location in the naming system. When this second object (the referee) is created and entered into the naming system, the referring object's weak reference is completed, at which point it is able to make full use of the facilities of the referee. Weak references thus remove the need to carefully deploy objects in a fixed order, and allows the use of circular object references, as an object may now be deployed before the object(s) it requires access to.

The naming system provides a single unified deployment space for all application content and functionality. The naming system thus resembles, to some extent, a more restrictive form of Lua's tables [128]. Alternatively, the framework's naming system also mimics the hierarchical nature of

XML's syntax. These similarities are intentional, and aim to support potential mappings from both Lua and XML to Fluid-based compositions.

Configuration System

In the context of the Fluid framework, a *configuration* is an XML description of a Fluid concept. XML configurations are used throughout the Fluid framework for the description of its various concepts, and will typically be used to populate the naming system or parameterise Fluid's runtime objects.

Fluid configurations are supported by the XmlCfg class, which corresponds to XML elements, and Values representing XML attributes. In order to represent a given XML element's child elements and attributes, each XmlCfg object stores an associative container mapping element and attribute names to their matching Value and XmlCfg instances. The resulting hierarchy of objects supported by this representation is illustrated by Figure 3.4.

Prior to use, a given configuration will be validated against its corresponding XML schema, and then parsed to form an appropriate XmlCfg and Value hierarchy. Certain parts of Fluid configurations may also undergo additional processing before they are parsed, in order to support concepts such as OODDP inheritance relationships (see below).

The configuration system provides a number of operations for parsing and manipulating XML configurations. For example, the configuration system encapsulates the functionality required to locate an XML file given a suitable filename, validate the file against its XML schema[8], and construct the XML file's corresponding XmlCfg representation.

Fluid's configuration system also employs the type and naming systems to parse a small selection of XML elements: it is able to construct runtime objects corresponding to Fluid's primitive and manipulation types. The configuration system is also able to construct namespace instances, and can thus form simple namespace and value hierarchies.

Finally, the configuration system is also responsible for supporting OODDP type inheritance relationships via a manipulation of the types' XML configurations. The inheritance manipulation operation takes two OODDP type or instance configurations, given as XmlCfg instances, and returns a new XmlCfg as its result. The pseudocode for Fluid's OODDP inheritance is given by Listing 3.2;

[8]Fluid's configuration system uses Xerces-C++ from the Apache Software Foundation for XML schema validation.

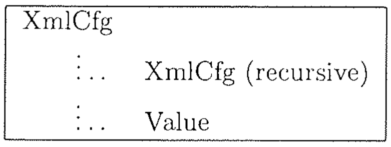


Figure 3.4: Representing XML configurations via Fluid's XmlCfg and Value classes.

the following provides an overview of the process while referring to line numbers in Listing 3.2 for completeness.

- The first step in this operation is to assign the parent configuration to the operation's result, giving result a copy of the parent configuration's content (Line 31). The result thereby *inherits* the parent configuration.
- Next, the parent configuration is both overridden and extended by the child configuration in the same recursive process. This process applies to both XML attribute and elements, which are collectively referred to as *items* in the following description for clarity. For each item in the child configuration, the parent configuration is checked for a corresponding item with the same absolute location and id attribute (for elements) or name (for attributes).
- If the parent configuration contains a corresponding item, then the child's item will *override* that of the parent. For XML attributes, overriding is performed by overwriting the attribute in the result with the child attribute's value (Line 5). Meanwhile, XML elements are *implicitly overridden* if the same element is held by both parent and child configurations: child elements are recursively processed in order to further apply overriding and extending manipulations. This is achieved via a recursive call (Line 15).
- If the parent configuration *does not* contain a matching item, then the child configuration must *extend* the parent: this is performed by inserting this item into the appropriate position in result (Lines 5 and 21).

The result of this process is a child OODDP type or instance configuration that inherits its parent's XML elements and attributes, with the exception of those items that it overrides or additionally provides. Once the appropriate inheritance manipulations have been applied to a given OODDP type description, the resulting configuration may be passed to the type system to be stored as a new OODDP type, or to Fluid's component model for the creation of an OODDP instance. Fluid's composition language tier is responsible for driving the processing of XML configurations, as well as their subsequent use by the framework (see Sections 3.2.2 and 3.2.3).

The Fluid framework's configuration system reintroduces much of the scene system's functionality for describing scene content. However, the configuration system's behaviour is better encapsulated, and defers certain operations to Fluid's component model. Whereas the prototype's scene system was responsible for the description *and* instantiation of OODDP types, the framework's configuration system is exclusively concerned with lower level XML manipulations, regardless of their higher level semantics.

Listing 3.2: The pseudocode for OODDP inheritance manipulations.

```
void processInheritance( XmlCfg& parent , const XmlCfg& child )
{
        // override or extend parent's attributes with those from child
        for_each attribute a in child
        {       parent[a] = child[a]
        }
        // override or extend parent's elements with those from child
        for_each element e in child
        {
                if parent[e] exists then
                {
                        // recursively call processInheritance in order to
                        // perform the necessary inheritance manipulations
                        // for sub-element e in both parent and child
                        processInheritance ( parent[e], child[e] )
                }
                else
                {
                        // assign a deep copy of child[e] to parent[e],
                        // thus extending parent with element e from child
                        parent[e] = child[e]
                }
        }
}

XmlCfg parent = ... // obtain parent type description from XML configuration
XmlCfg child  = ... // obtain child type description from XML configuration

// result is initialised to a deep copy of the parent type; the following assignment
// is equivalent to result inheriting every element and attribute from the parent type
XmlCfg result = parent;

// begin recursively processing result and child in order to perform any necessary
// overriding and extension of attributes and elements
processInheritance( result , child );
```

Furthermore, although the prototype's scene system was also able to generate value hierarchies from XML, the availability of the type and naming systems allows the framework's configuration system to make a number of improvements. For example, while the Fluid prototype used XML schema to validate data prior to use, the type information maintained by Value objects allows for continual run-time type and integrity checking. Meanwhile, the concepts offered by Value manipulations support an extensible range of complex configuration data to be described. The semantics of such manipulations are also retained by the type system, so that an `Optional<boolean>` does not lose information by becoming a `true` or `false` value in the data. These additional features support a more continuous relationship between configuration data and its eventual use in the naming system, OODDP instances and components.

3.2.2 Component Model

The Fluid framework's component model is responsible for both supporting and enforcing the use of *Fluid components* to define application functionality. Consequently, Fluid's component model provides a range of standards, XML schema, and operations for the definition of Fluid components, their deployment as part of Fluid applications, the specification of their interfaces, and the lifetime of component instantiations.

Fluid Component Definition

Fluid components conform to the definition of components as given in Section 2.2.3: they are units of composition with contractually specified interfaces and explicit context dependencies only. Furthermore, although Fluid components can be deployed to Fluid's naming system independently as part of OODDP Instances, they are ultimately subject to composition by third parties in order to form more complex Fluid applications.

More specifically, Fluid components are finely grained objects that will form *part* of higher level OODDP type and instance definitions. A number of components may be required to fully describe a given OODDP type, with each Fluid component encapsulating the functionality of a single *facet* of that type's conceptual representation. As a result, Fluid components will typically encapsulate functionality at the granularity of a small number of tightly coupled functions or events. The Fluid framework's use of fine grained components has been greatly influenced by that seen in data driven game engines. Also, while mainstream component based applications typically make use of class based components, a number of studies have explored the potential usefulness of components encapsulating smaller concepts [120, 119, 118].

Each Fluid component is an encapsulation of runtime behaviour and computation. In order to define complete Fluid applications, components must be composed and thus allowed to collaborate. Component composition is facilitated by the presence of bottleneck interfaces, which consist of both required and provided events and functions. The interconnection of bottleneck interfaces is described below. In addition, each component may also be parameterised using XML configurations. Component parameterisation will typically influence the component's runtime state and behaviour, leading to increased flexibility and implementation reuse. Finally, components may expose certain properties to the naming system, and therefore other components, by including named Values in their associative containers. An overview of a Fluid component, including the main elements given here, is depicted in Figure 3.5.

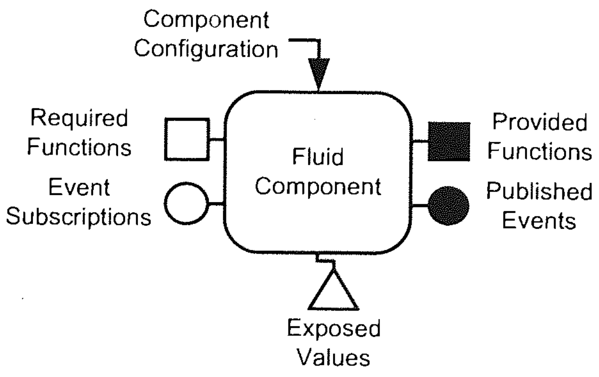


Figure 3.5: A Fluid component, including its bottleneck interface.

Fluid Component Deployment

In the context of the Fluid framework's component model, a component *deployment* corresponds to a single component representation, available at a known location, that may be used by the Fluid framework. Each Fluid deployment consists of two distinct parts providing separate component description and dynamically linked implementation. This approach differs from many component frameworks, such as those based on JavaBeans, where there is little or no distinction between implementation and description: in many cases, especially when Java is employed, a component's description may often be derived from its implementation using metadata or reflective functionality provided by the component itself.

However, C++ lacks the facilities required to easily access such information from Fluid components. A naïve solution *could* incorporate both implementation and description within the same component DLL, providing separate interfaces for introspection and component instance management. However, this approach would hide potentially useful information behind a C or C++ interface, placing additional and unnecessary requirements on any client wishing to access component description data. Instead, Fluid encapsulates component implementations using DLL files, and provides component descriptions using separate XML files. Fluid's component descriptions are thereby readily accessible to tools and casual browsing, and clients are only required to interact with DLL interfaces when making use of component implementations. Further wrapping for component distribution is easily facilitated by a wide range of third party Zip, Rar and Tarball file manipulation utilities.

The Fluid framework assumes the availability of one or more local repositories of Fluid component deployments. Note that *local* in this context refers to a local or networked directory accessible via a drive letter and absolute path. The Fluid framework places no restrictions on the number of component repositories, nor on the number of component deployments available in each repository. However, the framework assumes that both repository and deployment locations are known prior to executing any Fluid application, and that any absolute paths to such locations will remain valid during the application's execution. As component repositories are not the focus of this work, any further functionality or complications regarding their use are not considered by the Fluid framework's design or implementation.

Fluid Component Interfaces

Each Fluid component deployment includes a component description, or *specification* file, which is written in XML and must conform to an XML schema provided by the Fluid framework. The specification file is responsible for describing the bottleneck interface of a single component, as well as its configuration requirements. Each component specification includes the following items:

- An `id` attribute, in the root of the XML file, which uniquely identifies the deployment.
- A `location` attribute, in the root of the XML file, which gives an absolute path declaring the location of the component's DLL. Note that the component's specification and implementation are not required to be stored in the same location.
- An `Exports` element containing a description of the functions and events *provided* by the component. Each export is described as a function or event signature, and will be converted into a corresponding signature type description at runtime by the type system.
- An `Imports` element, which contains a description of the functions and events *required* by the component. As with the `Exports` element, each signature description will be converted and used at runtime by the type system.
- A `Configuration` element, which contains a list of basic and manipulation type descriptions. Each type description under the `Configuration` element will be converted to a runtime type object, and the resulting sequence of type objects will be used to validate the component's configuration data. Component configuration is described below.

All Fluid components are black box objects that expose their functionality via bottleneck interfaces. A Fluid component's bottleneck interface describes the functionality provided by the component, as well as the component's contextual dependencies. The functionality of Fluid components is not restricted by a fixed interface, but is instead described by the component itself: each component is free to provide and require both functions and events, as described below.

A Fluid component exposes free, static or member functions using the `Function` class provided by the type system (see Section 3.2.1). All exposed functions are deployed alongside their owning component instances, as child entries in the naming system, regardless of their initial scope or ownership. That is, all function exposures are treated as members of the component providing them. Subsequent to exposure, a function may be accessed by other components in the naming system's namespace hierarchy using a weak reference as described in Section 3.2.1. Function references define a functional operator that allows the references to be invoked as if they were locally declared function pointers. Their use by referring components looks like any other function call.

Components may also expose (or *publish*) functionality using the type system's `Event` class (see Section 3.2.1). Event signatures do not specify a return type, but may include a single optional *payload* parameter. Like functions, published events are deployed in the naming system as child elements of the component exposing them, and may be referenced (or *subscribed to*) by other components in a similar way. However, unlike function referencing, subscribing components must also provide an *event* *handler* function in order to create a complete event reference. At runtime, the publishing component will *trigger* the event, which will in turn call subscribing components' event handlers, passing the event payload (if any) as the call parameter.

Fluid's function and event concepts support two distinct methods for inter-component communication and collaboration: `Function` provides a single cast message passing concept, which transfers control from a single calling component to a single called component. By contrast, `Event` supports multicast message passing, transferring control from a single *triggered* publisher component to multiple subscriber components. Although both `Function` and `Event` could have been implemented using a lower level message passing concept, their distinctive availability in the Fluid framework allows for a more terse and explicit description of the intended application control flow. Figure 3.6 shows an example components topology, with a given component instance exposing both events and functions with multiple connections.

As described in Section 3.2.1, `Function` and `Event` signatures must define their return value and parameter types using Fluid's basic types and manipulations. The Fluid framework provides a number of mappings from its basic types to a number of standardised C++ types; `Functions` with no return value and `Events` with no payload may use `nil` in place of the C++ `void` keyword.

Fluid Component Lifetime

Deployment DLLs are required to provide two functions for the management of component lifetimes: a `create` function, which is responsible for constructing component instances; and a `destroy` function, which destructs component instances. Deployment DLLs are thus implementations of the Factory Method pattern [132], with each DLL providing a single concrete creator. The Fluid framework implementation ensures that components are created and destroyed by the same deployment DLL.

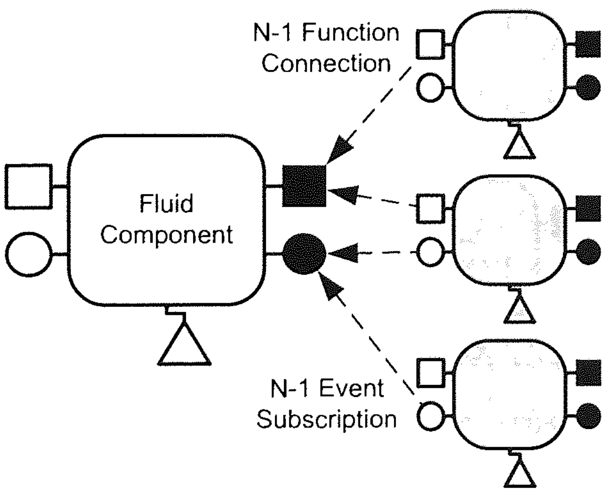


Figure 3.6: An illustration of how components are connected via their bottleneck interfaces.

When invoked by Fluid's component model, a deployment's create function is given two parameters: the first is an XmlCfg instance containing a list of attributes, which may be used to parameterise component instantiation. The second parameter is an instance of the `NamespaceAccess` class, which is responsible for providing controlled access to the naming system's contents. The relationships between a Fluid component, its create function parameters, and its deployment specification are shown in Figure 3.7.

A deployment's create function is given an XmlCfg parameter containing a list of Value instances, which may be used to configure the component instance it returns. The list of Values may consist of basic types and manipulations, which must conform to a corresponding list of Value name and type descriptions given in the component's specification. A component configuration that has been validated against the corresponding deployment's component specification may be used by its create function during the construction of a component instance.

Component configuration is an important part of Fluid framework, and can potentially increase implementation reuse by allowing a wide variety of OODDP instance behaviours to be defined with a small selection of component deployments. Furthermore, component configurations form part of OODDP type and instance descriptions, and may thus take part in OODDP inheritance relationships. It is therefore possible to describe a range of OODDP instances with varying configurations, without having to describe each instance entirely: application developers can define an OODDP type describing a default configuration, and subsequent OODDP instantiations need only include those elements that differ from this default. OODDP inheritance manipulations will ensure that the resulting instance descriptions inherit the default configuration, while overriding or extending the default configuration with those elements and attributes particular to each instance.

The Fluid framework is responsible for enforcing that a deployment's DLL constructs components that conform to the deployment specification. This is done by placing restrictions on what the com-

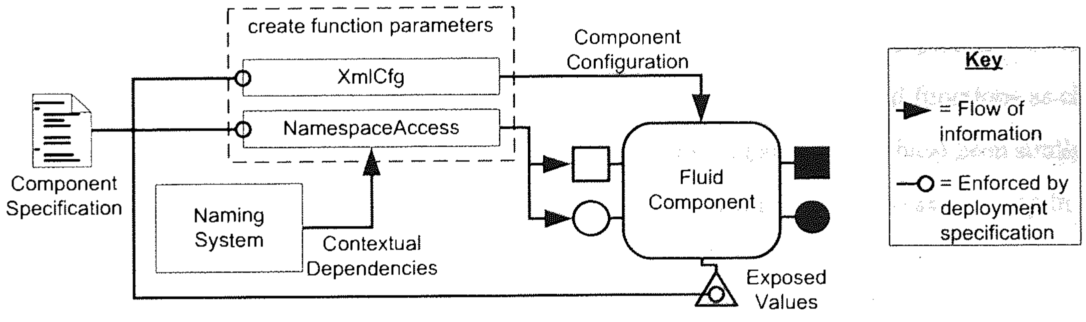


Figure 3.7: Creating a parameterised Fluid component. Each component imports its contextual dependencies, while optionally exporting any number of Value instances. A component's interactions with the Fluid framework, including its imports, exports and configuration, are validated against the component's specification.

ponent is allowed to write to and read from the naming system during its construction. Firstly, each component must write at least its specified exports to child elements in its own associative container. This restriction guarantees that component instances provide the minimum functionality promised by their specifications. Also, each component may only access those namespace entries corresponding to the import entries in their specifications. The latter restriction is enforced by the `NamespaceAccess` class, which provides the only means by which a deployment's create function may access the naming system's contents. Limiting a component's namespace access to its predefined imports precludes component implementations with greater contextual dependencies than those specified.

As well as accessing its specified namespace imports, a component also has access, via the `NamespaceAccess` class, to its enclosing OODDP instance object in the naming system. An OODDP instance's namespace entry may contain any number of component instances; these instances collectively represent the functionality of the OODDP instance, and are analogous to object methods. The OODDP instance object available via the `NamespaceAccess` class of component instantiation is therefore equivalent to the `this` object in languages such as C++ and Java, and the `self` object in Lua. In a similar way to such languages, all components have access to the associative container of their enclosing OODDP object, and thus its Values and other components. This feature allows OODDP instances to behave more like their counterparts in object oriented implementation languages.

By validating a component's described bottleneck interface against its interactions with Fluid's naming system contents during instantiation, Fluid's component model is able to enforce that runtime component topology is consistent with what is defined by application descriptions. Furthermore, by disallowing additional inter-component dependencies, Fluid's component model is able to fully understand the current relationships between component instances. This understanding could facilitate optimisations such as the removal of unreferenced components, and potentially the introduction of garbage collection to Fluid's composition language.

Fluid components created via component deployments will be constructed by their corresponding component factory with a given configuration. During its instantiation, each component will form collaborations with other component instances by assigning its exported events and functions as child entries in its own associative container, and by importing functions and events that have been similarly exported by other components. Once instantiated, the component will be accessible as an entry in the naming system; the actual placement of the component instance will depend on the application configuration currently in use. During application execution, a component's runtime behaviour is invoked by other components, either by directly calling the component's exported functions, or triggering events to which this component has subscribed. When no longer required, the component will be passed to its deployment's destroy function, which is responsible for calling the correct object destructor and restoring memory to the heap. Fluid's `Function` and `Event` objects will automatically disconnect themselves as part of the component's destruction, and thus sever any collaborative connections the component had during its lifetime.

3.2.3 Composition Language

The Fluid framework builds upon the functionality supported by its lower tiers in order to provide an XML based composition language. Fluid's composition language provides a textual interface for describing component applications via the selection, connection and configuration of Fluid components. Each component application is described by an XML *application description* document, and Fluid's composition language is responsible for supporting the use of application description documents to define component applications. In order to meet this requirement, the composition language tier supplies a number of XML schema that define the syntax for application descriptions, as well as a range of operations that open, parse and process application descriptions in order to drive lower level functionality.

Fluid's composition language has a declarative syntax that drives a range of runtime activity in the framework's lower tiers. Application descriptions are parsed in order to populate the naming system with a hierarchy of Values, OODDP types and instances, and Fluid components. Subsequent runtime behaviour is provided by the implementation of Fluid components, and the framework's component model. Like many other XML based composition languages, Fluid's composition language provides the means by which an *initial* application configuration may be specified. The reconfiguration of software structure is facilitated by certain operations available via Fluid's component model.

The Fluid framework provides an number of XML schema that collectively stipulate the hierarchical structure of application descriptions. An overview of the structure described by the schema is given in Figure 3.8; the concepts supported by this structure are discussed further below.

The concepts illustrated by Figure 3.8 correspond to equivalent concepts maintained by the naming system's hierarchy of associative containers, as seen earlier in Figure 3.3. Consequently, those XML elements introducing such concepts must provide an `id` attribute, so that the object stored by the naming system may be identified at runtime. Certain concepts, such as the events and functions exposed by Fluid components, are required to refer to other concepts described in application descriptions; these inter-concept references may be made using both relative and absolute paths making use of the identifiers specified by `id` attributes.

Namespace Namespaces in an application description have a direct correspondence with those in Fluid's naming system. When parsing the application description, each Namespace element is created as a namespace instance. Elements between the opening and closing Namespace

- Namespace
  - Namespace (recursive)
  - Value
  - OODDP Type
    - (As OODDP Instance)
  - OODDP Instance
    - OODDP Instance (recursive)
    - Value
    - Component
      - Import
        - Event
        - Function
      - Configuration
        - Value

Figure 3.8: The hierarchical structure of Fluid's composition language.

elements are further parsed to become children of the namespace instance. Each application description must begin with a `root` element with the `root` identifier, which corresponds to the root namespace in the naming system.

Namespaces allow a Fluid application developer to organise software structure into associative groups, which help to avoid ambiguity and may also be used to represent concepts such as ownership, authorship, distinct packages or libraries. Fluid's use of namespaces is motivated by similar concepts in C++ and Lua. Like these concepts, Fluid's namespaces are *open*, in that they may be added to after the point of their initial description.

Value Values in application descriptions generally correspond to variables used in implementation languages, although their particular meaning depends on where a given Value resides in an application description. Values contained within opening and closing `Namespace` elements correspond to global variables, while those forming part of an OODDP Type or Instance description (see below) resemble class and instance attributes respectively. Finally, Value descriptions forming part of Fluid component descriptions (again, see below), or within the `Configuration` part of Component descriptions may be considered part of the components' configuration and parameterisation. Regardless of its location, each `Value` element in an application description is parsed and a corresponding `Value` instance is created.

All `Value` elements specify both a `Type` *and* `Data` part in order to describe the Value's type as well as an initial assignment. The XML schema for Values allows a range of type information and data assignments to be described, including primitive values and complex recursive manipulations. At runtime, a `Value` capable of supporting the described type is first constructed, and then the data part of the Value description is assigned to initialise the `Value` instance.

The verbose description of Values is used to increase type correctness and avoid ambiguity: a reduced syntax requiring only an initial assignment could stipulate a Value containing the integer `15`, but actually representing an `Optional<int>`, a `Sequence<int>` containing one item, and so on. Although a more verbose syntax for Value descriptions leads to more complex and longer XML documents, the additional information held by a Value description also helps to retain the semantics of the initial description, and leads to greater type safety. For example, an application developer may model the triplet {`yes`, `no`, `unknown`} using an `Optional<boolean>`, which will only accept `true`, `false` and `nil`, respectively, throughout the lifetime of the `Value` instance. While this representation does not directly correspond to the developer's intentions, the `Value` instance maintains more of the initial meaning than a naked boolean holding only `true` or `false`.

OODDP Type An XML OODDP Type description consists of both opening and closing `Type` elements, with an `id` attribute to name the OODDP Type, and an optional `parentType` attribute to name a parent OODDP Type description. Everything between the opening and closing `Type` elements are part of the described type, and may include descriptions for `Values`, `Components` and nested `Instances`. Subsequent to being parsed by the Fluid framework, OODDP Type descriptions are maintained as static textual representations in the Fluid framework. As such, OODDP Type descriptions are similar to classes in OO languages. OODDP inheritance is supported as a manipulation over OODDP Type descriptions: the XML elements of a parent type description may be inherited, overridden or extended by a child type description.

OODDP Instance OODDP type descriptions may be instantiated by Fluid's component model in order to form OODDP `Instances`, which correspond to dynamic OO objects. Although Fluid's `Instance` class is merely a `Namespace` specialisation with additional methods exposing OODDP Type information, the recursive instantiation of OODDP Types will typically include the construction of `Values`, `Components`, and their constituent parts, which contribute to overall application state and behaviour. As such OODDP Instances collectively *define* application behaviour, using object oriented abstractions to guide the granularity, separation and identification of software structure.

Each OODDP Instance is described by a pair of opening and closing `Instance` XML elements. The opening element must contain an `id` attribute to name the OODDP Instance, just as an identifier would be associated with an implementation language variable. In addition, a `parentType` attribute may be provided, in which case the Instance description will take part in OODDP inheritance manipulations with the given parent type prior to its instantiation; the Instance description is free to include Value, Component and nested Instance descriptions in order to override or extend those described by the parent OODDP Type description. If no `parentType` attribute is given, then the instantiation process only considers the given Instance description. Regardless of the presence of a `parentType` attribute, everything between the Instance's opening and closing elements is considered part of the Instance description.

Component Fluid component implementations are subject to composition by third parties in order to collectively provide the custom behaviour of many different Fluid applications. In order to achieve this, component elements of application descriptions are required to provide the *selection*, *interconnection* and *configuration* of Fluid components.

Each component description consists of a number of XML attributes and elements provided between opening and closing `Component` elements. The opening element must include an `id` attribute, which identifies the component's entry in the naming system. Furthermore, a particular Fluid component is *selected* for composition by providing a `type` attribute as part of the opening `Component` element. The identifier given by the `type` attribute must correspond to the name of a component deployment available to Fluid's component model (see Section 3.2.2). At runtime, Fluid's component model will use the component deployment identified in order to further process and instantiate a given Component description. Those XML elements lying between the opening and closing `Component` elements will be parsed as part of the process of component instantiation.

As described in Section 3.2.2, each component deployment supplies a specification stipulating the name and type information of its component's bottleneck interfaces. Both provided and required functions and events are defined. These bottleneck interfaces are *connected* by the `Imports` element forming part of each Component description.

The `Imports` element contains a number of `Event` and `Function` elements, each of which consists of an `id` attribute and either a `exportedEvent` or `exportedFunction`. The `id` of such imports must match the `id` attribute of a corresponding `Event` or `Function` element in the `Imports` section of the component's specification document. The `id` attributes in the component specification document and Component description thus connect the type information of an import (in the specification document) with its instantiation configuration (in the Component description). The `exportedEvent` or `exportedFunction` attribute of an opening `Component` element gives the relative or absolute path to an `Event` or `Function` instance in the naming system.

During its runtime instantiation, each Component will create a number of `Event` and `Function` instances, as described by the `Exports` element of its specification document, that collectively form a bottleneck interface to its implementation details. These `Event` and `Function` instances will be added as children of the `Component`'s instance in the naming system hierarchy. During the same instantiation process, each Component will also attempt to import the Events and Functions described by the contents of its `Imports` element. Each imported Event or Function will be located via the path given by the import's `exportedEvent` or `exportedFunction` attribute, and the type of the `Event` or `Function` instance will be checked against the import's type information as given by the component's specification document. If the type information does not match, then Component construction will fail.

Finally, the `Configuration` element of a Component description contains a sequence of `Value` elements providing component *configuration* and parameterisation. The `Value` elements described by a Component configuration are parsed in the same way as all other Value descriptions in order to generate corresponding `Value` instances. However, once created, the type of each Value is checked against an element with the same `id` attribute in the component specification's `Configuration` section. For each `Value` element in the Component description, the component specification must contain an entry with the same identification, and describing the same type as what is given by the current `Value` element. If a corresponding Value specification is not found, then the Component will not be instantiated. Meanwhile, Component configurations that pass this validation test will be supplied as part of Component instantiation, and may be used to configure the resulting `Component` instance.

Fluid's composition language brings together the concepts of composition languages from mainstream academia and industry and OODDP from game development, and thus introduces a novel object oriented abstraction for the definition of software compositions, making use of a range of concepts at the scale of OO software structure. The resulting composition language exhibits a number of object oriented features: Fluid's `Values` and `Components` respectively provide attribute and methods abstractions, while static OODDP `Type` descriptions support an analogy of OO classes. OODDP `Instances` are dynamic representations of OODDP Types, and thus correspond to objects in object oriented terminology.

While Fluid's OODDP implementation supports a number of OO concepts, the motivation for their incorporation is not to produce an OO composition language, but to develop a composition language where certain OO concepts introduce additional expressiveness and accessibility for the definition of flexible software structures using fine grained, highly configurable components. Accordingly, Fluid's composition language supports OODDP inheritance as a first step toward bridging the clear disparity between the design of traditional OO and component based software [119]. However, Fluid's OODDP implementation currently lacks certain fundamental OO concepts such as encapsulation, which are often present in the majority of OO implementation languages[9], but have not been considered by Fluid's design thus far due to a foremost focus on software composition and configuration.

3.2.4 Fluid Executable

The Fluid executable is responsible for bringing together the three tiers of the Fluid framework, which have been implemented as a number of C++ static libraries, to form a single coherent component framework capable of supporting a range of potential Fluid component applications. The Fluid executable represents the entry point for all Fluid applications, and provides the means by which all Fluid applications will be launched. Fluid's executable is therefore responsible for the overall control flow and lifetime of all Fluid applications, as well as for the Fluid framework itself. In order to meet this requirement, the Fluid executable performs a sequence of high level actions, as discussed below.

1. **Enter Fluid executable** The Fluid executable begins its lifetime as a Windows process, whereupon control flow passes to its `main` C++ function.

2. **Parse command line** The Fluid executable accepts a number of command line parameters, which collectively indicate which Fluid application is to be launched. An `app_config` option must be provided in order to specify the location of an application description file. Following this, a list of `component_spec` parameters may be used to give the relative or absolute paths of one or more component specification files. Alternatively, an optional `config` parameter can be used to direct the Fluid executable to a command line configuration file, which may contain additional `component_spec` and `app_config` parameters. The use of a command line configuration file is recommended for more complex Fluid applications making use of many different component deployments.

3. **Create Fluid framework** Once the executable's command line has been parsed, the Fluid framework is created by instantiating one or more objects corresponding to each tier: a `TypeManager` instance maintains OODDP types on behalf of the type system, while a `Namespace` instance provides the root namespace for the naming system. Meanwhile, a `ComponentFactory` instance encapsulates the majority of Fluid's component model functionality. There is no single class corresponding to the configuration system or component model; their operations are invoked via façade interfaces comprising related functions in appropriate namespaces.

[9]For example, C++ has the `public` `private` and `protected` keywords to control access rights, and thus the degree of encapsulation, for class methods and attributes.

4. **Register components** Fluid components are registered with the framework by parsing each component specification given as a command line parameter, obtaining bottleneck interface and configuration information from the specification, and then linking with the deployment's DLL in order to locate its create and destroy functions. Each component used in the application description *must* be registered in this way before the application description itself is parsed. If an unregistered component is named in the application description, then the Fluid framework will not be able to locate or verify that component, and instantiation of the Fluid application will fail.

5. **Deploy System instance** Once Fluid's component framework is ready to instantiate a Fluid application, an OODDP instance named *System* is inserted into the root namespace. The System instance is responsible for exposing a range of essential functionality to Fluid components, and thus performs a role similar to that of the standard libraries in Lua. Unlike the creation of OODDP instances and Fluid component described elsewhere, the process of inserting the System instance is not restricted by bottleneck interface or configuration checks, as the instance is inserted by the Fluid executable itself. Indeed, many of the operations provided by the System instance would not be possible if they were not exposed in this way. The functionality exposed by the System instance includes the following:

   Application loop The ApplicationLoop component is responsible for encapsulating the concept of the application loop. The ApplicationLoop component exposes an `update` event, as well as two functions allowing other components to stop the application loop and check if the application loop is currently active. The Fluid framework is required to trigger the application loop's update event, allowing its subscribing components to provide a wide range of runtime behaviours. The Fluid framework will repeatedly trigger the update event until the ApplicationLoop's `stop` function is called, in order to produce the illusion of continuous functionality.

   Namespace access The NamespaceAccess component's intended use is for monitoring the current status of the naming system and its constituent elements. In order to support this functionality, the NamespaceAccess component exposes a generic `visitRootNamespace` function, which takes a visitor object capable of operating on the naming system's contents (see Section 3.2.1). The visitor passed to visitRootNamespace is initially given immutable access to the root namespace, although it is possible to visit all elements of the naming system by selectively recursing on associative containers. An overload of the `visitRootNamespace` function takes an absolute path, and providing the named element exists, will pass the named element to the given visitor in place of the root namespace.

Namespace management The NamespaceManagement component exposes a number of functions for modifying the contents of the naming system. Three functions are exposed at present: one for creating OODDP instances and inserting them into the naming system, another for moving them from one namespace or parent OODDP instance to another, and a final function for destroying an OODDP instance at a given location. The NamespaceManagement component supports a number of experimental features, allowing for the dynamic reconfiguration of application descriptions. The functions exposed by the NamespaceManagement component are intended to allow application developers to manipulate OODDP instances during runtime execution, and further emphasise the use of OODDP types and instances as the main unit of application description, deployment and manipulation.

6. **Parse application description** Once the System instance has been inserted into the root namespace, the Fluid framework parses the application description file. Each element of the application description is processed, with corresponding objects being created and inserted into the naming system as described in Section 3.2.1. Once the entire application description file has been parsed and all namespace entries inserted, the Fluid application is ready for execution.

7. **Enter application loop** Fluid's application loop is implemented by the Fluid executable, which first obtains a handle to the System instance's application loop component, then continuously triggers its update event while the application is running. The Fluid executable implementation is illustrated by the following pseudo code:

```
// appLoopComponent is a handle to the System instance's application loop component
while( appLoopComponent->isRunning() )
{    appLoopComponent->triggerUpdateEvent();
}
```

Each time the application loop component's update event is triggered, those components subscribing to the event will also be triggered. As control flow passes to component implementations, each component will be able to perform its own operations, potentially propagating control flow down to further components via event triggers and function calls. In this way, the application loop component's update event is the source of all runtime functionality for Fluid applications. The application loop, and thus all runtime behaviour, will terminate when a component implementation calls the application loop component's exposed `stop` function.

8. **Destroy root namespace** When control flow exits Fluid's application loop, the Fluid executable will destroy the naming system's root namespace. This results in the namespace system's contents being removed in depth first fashion, as each namespace instance will destroy

its contents prior to its own destruction. The Fluid executable thus dissolves the current Fluid application.

9. **Destroy Fluid framework** Once there is no longer a requirement to represent the current Fluid application, the Fluid framework itself may be destroyed. This results in all resources previously held by the framework being relinquished, including handles to component deployment DLLs.

10. **Exit Fluid executable** Finally, with all resources released, the Fluid executable terminates gracefully and its Windows process is destroyed by the operating system kernel.

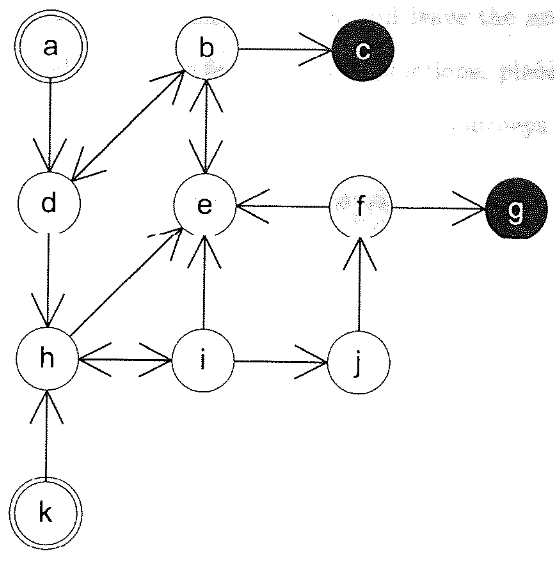
The Fluid executable provides a fixed skeleton or pipeline of runtime behaviour. Application specific functionality and flexibility are provided by application descriptions, which specify the selection, interconnection and parameterisation of data driven components. Section 3.3 provides a complete example within the context of the geovisualition domain.

3.3 Complete Example

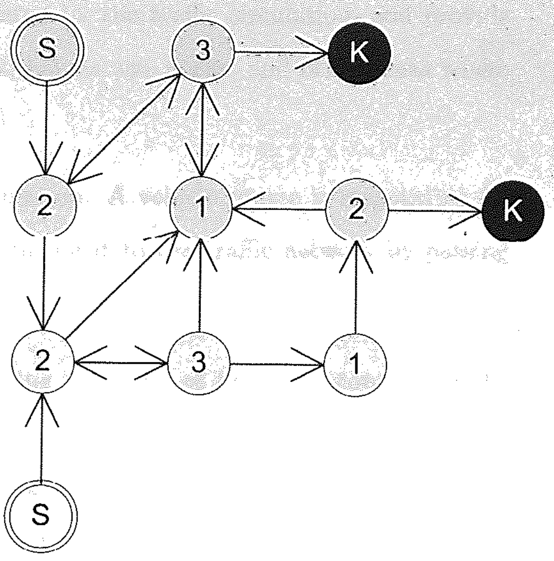
This section provides an example use of the Fluid framework to develop a simple traffic simulation represented by the network shown in Figure 3.9. The intention of this example is to demonstrate how OODDP can be used as an expressive reuse mechanism as part of an XML based composition language, while also providing a more concrete illustration of the Fluid framework's functionality as described in sections 3.2 through 3.2.4.

The simple example given in this section was chosen because it is able to demonstrate the successful combination of OODDP and component based concepts without superfluous complexity, and whilst remaining accessible to non experts. The traffic simulation employs OODDP manipulations to describe the simulation's constituent concepts, making use of OODDP inheritance, overriding and extension where appropriate. Meanwhile, the Fluid framework's component model both enforces and supports the use of a number of small scale configurable components that collectively define overall application functionality. Finally, a range of emergent behaviour exhibited by the executing simulation illustrates the successful combination of technologies in the Fluid framework: the example's various components and configurations cooperate at runtime to form a working, albeit simple, traffic simulation and appropriate real-time geovisualisation.

Section 3.3.1 presents an overview of the traffic simulation's high level design. The simulation's implementation is defined using Fluid component instances, component specifications, and an appropriate application configuration. Section 3.3.2 describes the Fluid component implementations and their specifications, while Section 3.3.3 presents the application configuration including the OODDP



(a) Junctions are represented as nodes in the traffic network



(b) The number of outgoing lanes for each junction in the traffic network shown in Figure 3.9a

Figure 3.9: The traffic network used throughout this example. Circles denote junctions, with doubly ringed circles representing vehicle sources and filled black circles representing vehicle sinks. Directed arrows correspond to lanes, indicating the direction of vehicle travel.

types and instances used to define the Fluid application. Section 3.3.4 illustrates the Fluid executable's runtime behaviour, building upon the material given in Section 3.2.4 with concrete details from the traffic simulation application. Section 3.3.5 presents the results of the traffic simulation, including a number of screen shots taken during its runtime visualisation.

The traffic simulation described in this section provides a complete application of the Fluid framework's functionality, the results of which may influence further developments, improvements and refinements of its design and implementation. To this end, Section 3.3.6 provides an evaluation of the traffic simulation application, highlighting the limitations of the Fluid framework's current design and implementation and discussing some suggested improvements.

3.3.1 Design

The traffic network topology shown in Figure 3.9 consists of junctions, lanes, vehicles, vehicle sources and vehicle sinks. The aim of the example traffic simulation is to support the following dynamic behaviour:

- The vehicles in this traffic simulation are minimal representations used to denote vehicle instances. Consequently, vehicles do not posses individual behaviour such as an intended destination or driving style, although a visual representation of each vehicle must be provided by the application.

- Vehicle sources and sinks represent the boundaries of the traffic network. They correspond to where vehicles will enter and leave the area considered by the traffic simulation, and provide abstractions for motorway junctions, places of work, homes, car parks, and other areas where vehicles may begin and end their journeys.

- Vehicle sources introduce vehicles to the traffic simulation. A vehicle source is responsible for periodically instantiating a vehicle object and introducing it to the traffic network by passing the instance to an outgoing lane.
- Lanes are unidirectional, with bidirectional roads being modelled as two lanes connecting the same two junctions but in opposite directions. Each lane is responsible for updating its vehicles in order to move them along until they reach the end of the lane. Lanes must maintain the ordering of incoming vehicles, so overtaking behaviour will not be supported, and the lane will also be required to prevent vehicles from hitting one another. Lanes must therefore take other vehicle positions into account when updating each vehicle. When a vehicle reaches the end of a lane, the vehicle is passed to the lane's outgoing connection, which will be a junction or vehicle sink.
- Junctions may have N incoming lanes and M outgoing lanes, and represent locations at which vehicles may change direction. A junction instance will accept one vehicle at a time from among its incoming lanes and pass the vehicle to one of its outgoing lanes. Vehicles are not permitted to wait on junctions, but are instead moved from an incoming lane to an outgoing lane as a single operation. Junctions with more than one outgoing lane will exhibit behaviour corresponding to junctions with traffic lights controlling traffic flow. Each junction will maintain runtime state dictating which outgoing lane is currently available to vehicles. Junctions will periodically cycle through their outgoing lanes. A vehicle arriving at a junction will be passed to the junction's currently available outgoing lane.
- All vehicles travel along the traffic network's lanes and junctions until they reach a vehicle sink, at which point they are destroyed and removed from the simulation.
- A real time visualisation of the traffic simulation's current state should be available, with vehicles, sources, sinks, junctions and lanes all rendered using appropriate representations.

The traffic network shown in Figure 3.9 has been designed to demonstrate a number of vehicle sources, sinks and junctions with varying numbers of incoming and outgoing lanes. The traffic network also includes both unidirectional and bidirectional roads, and supports a number of opportunities for looping routes. For example, a vehicle travelling to junctions at nodes `d`, `h`, `e` and `b` may return to the junction at node `d` rather than leaving the simulation via the vehicle sink at node `c`. Each concept in the traffic system is represented in the Fluid application by a distinct OODDP type, as discussed in Section 3.3.3. The following section describes a component based approach to representing the above behaviour.

3.3.2 Component implementations

In order to implement the above design, the OODDP types described above have been defined using the following component types.

Event propagation An `UpdateSimulationComponent` is used to control the high level behaviour of the traffic simulation by propagating the update event emitted by the Fluid framework's System instance, as described in Section 3.2.4. An `UpdateSimulationComponent` instance subscribes to the System instance's update event, but also exposes an update event of its own. Each update event received by the `UpdateSimulationComponent` is considered to represent the start of a new *time frame* in the traffic simulation. Fluid components constituting the various traffic simulation elements subscribe to the `UpdateSimulationComponent` instance's update event in order to be notified of each time frame as it occurs. The `UpdateSimulationComponent` forwards the Fluid framework's update event in this way so that it may introduce application-specific behaviour such as simulations with varying time scales, while also retaining control of the traffic simulation's runtime behaviour.

The `UpdateSimulationComponent` may be configured by specifying the time frame at which the traffic simulation should be terminated. The traffic simulation's runtime behaviour can also be sped up or slowed down by providing the duration or *delta* of each time frame. The update event exposed by `UpdateSimulationComponent` instances includes the frame delta as an event parameter, so that the current time scale is propagated to every element of the simulation.

Listing 3.3 gives the specification document for the `UpdateSimulationComponent`. The contents of Lines 1 to 7 provide the version and encoding of the XML document, as well as the identities of any XML namespaces used. These lines also provide the location of the `UpdateSimulationComponent` implementation DLL, and the location of the schema against which the specification document in Listing 3.3 should be validated. As Lines 1 to 7 will vary little between component specifications, they will be omitted from further examples for clarity.

Lines 9 to 15 of Listing 3.3 describe the functionality provided by the `UpdateSimulationComponent`, while Lines 16 to 28 describe its contextual dependencies. The `UpdateSimulationComponent` replaces Fluid's update event with its own scaled version as discussed above by importing a single event with no parameters, while exposing an update event with a parameter of type `float`. The `UpdateSimulationComponent` also exposes a function allowing clients to terminate the simulation at any time.

The contents of Lines 29 to 36 describe the `UpdateSimulationComponent`'s configuration. The purpose of `stopAtFrame` and `frameDelta` have been discussed above.

Junction behaviour Junctions in the traffic system represent a number of interconnected decision points, each with N incoming and M outgoing lanes. Junctions are responsible for forwarding incoming vehicles to an appropriate outgoing lane according to the current state of their traffic lights.

Each junction is modelled as an OODDP instance containing a component providing the junction's behavioural functionality. A junction may also contain a single vehicle instance corresponding to the vehicle being passed from an incoming lane to the currently available outgoing lane. The junction's behavioural component has access to nested vehicle instances via the *concept of self* that was provided during the component's instantiation.

A junction's behavioural component performs the following operation upon receiving an update event: if a vehicle is currently waiting to be processed by the junction, then the vehicle is passed to an outgoing lane according to the current state of its traffic lights. A junction's traffic lights will be represented as an enumeration indicating that incoming vehicles are able to proceed to outgoing lanes A, B or C, depending on the junction's number of outgoing lanes. A junction will modify the runtime state of its traffic lights by cycling through the outgoing lanes available to the junction. Each outgoing lane should be navigable for a parameterised duration, allowing the distribution of traffic flow to be controlled by the application's configuration.

The implementation of junctions within the Fluid framework is complicated by two restrictions. The first restriction is due to component instances only having immutable access to the root namespace, as described in Section 3.2.2. Consequently, vehicle instances cannot be passed between junctions and lanes directly, but must instead be moved using the `moveInstance` function provided by the Fluid framework's System instance. This limitation presents two possible ways of moving vehicles, based on push and pull semantics.

- The push method would rely on junctions to pass vehicle instances to outgoing lanes using the `moveInstance` function. In order to facilitate this, each junction would require the paths of its outgoing lane instances as part of its configuration. Junctions should also inform outgoing lanes of any added vehicles, and so lanes must provide a `vehicleAdded` function that may be called by junction instances.

- The pull method would be based on junctions emitting a `takeVehicle` event indicating that a vehicle with a given id must be moved to an outgoing lane. An outgoing lane would subscribe to and subsequently receive the event, and invoke the `moveInstance` function with the vehicle

Listing 3.3: The specification document for the UpdateSimulationComponent component.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Component id="UpdateSimulationComponent"
  location="X:\TrafficSimComponents\UpdateSimulationComponent.dll"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="fluid_namespace"
  xsi:schemaLocation="fluid_namespace
    X:\libraries\fluid\fluid\schema\component_specification.xsd">

  <Exports>
    <Event id="onUpdate">
      <Type>
        <Payload xsi:type="PrimitiveType" typename="Float"/>
      </Type>
    </Event>
  </Exports>
  <Imports>
    <Event id="onUpdate">
      <Type>
        <Payload xsi:type="PrimitiveType" typename="Nil"/>
      </Type>
    </Event>
    <Function id="stopApplication">
      <Type xsi:type ="FunctionTypeNode">
        <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
        <Parameters/>
      </Type>
    </Function>
  </Imports>
  <Configuration>
    <Value id="stopAtFrame"><!-- -1 == NEVER_STOP -->
      <Type xsi:type="PrimitiveType" typename="Integer"/>
    </Value>
    <Value id="frameDelta">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Value>
  </Configuration>
</Component>
```

id in order to take ownership of it.

The second limitation of a Fluid based approach is that component specifications must declare all provided exports and contextual dependencies individually and explicitly so that they may be validated against the application configuration. For instance, a junction with N incoming lanes and M outgoing lanes must explicitly describe N connections for the pull method in order to subscribe to N `takeVehicle` events, or M connections for the push method in order to be able to call M `vehicleAdded` functions.

In addition, each enumeration of incoming or outgoing lanes would require its own component implementation and specification. The pull method would require N distinct junction implementations in order to subscribe to N event sources, but would allow for any number of outgoing lanes. Meanwhile, the push method described above would require M distinct junction implementations to support M outgoing lanes, but would allow for any number of incoming lanes. Finally, event connections in the Fluid framework are described using the path to the OODDP instance publishing the event, rather than relying on a global message broadcast system.

Due to the above limitations of the design and implementation of the Fluid framework, the traffic simulation application makes use of the push method of passing vehicles between lanes and junction instances, in order to avoid too many explicit paths in the application configuration. An improved event system based on event types and global event broadcast would support more efficient event subscriptions and application configurations.

The traffic network shown in Figure 3.9 requires junctions with one, two and three outgoing lanes. Consequently, three junction components have been implemented, and named NtoMVehicleManagerComponent, where M is replaced with 1, 2 or 3 according to the number of outgoing lanes supported by each component implementation. Listing 3.4 gives the specification for the `Nto3VehicleManagerComponent`. The `Nto1VehicleManagerComponent` and `Nto2VehicleManagerComponent` are similarly specified, except that the import and configuration elements for lanes B and C are not present in the specification for the `Nto1VehicleManagerComponent`, while those for lane C are not present in the specification for the `Nto2VehicleManagerComponent`.

`Nto3VehicleManagerComponent` exports the `handoverVehicle` function, which allows incoming lanes to inform the junction that a vehicle instance has been added. `handoverVehicle` takes the id of the vehicle added as its single parameter. The `isSpaceAvailable` returns whether the junction is able to receive an incoming vehicle. `spaceAvailableOnOutgoingA`, `B` and `C` are imported to allow the junction to query its outgoing lanes to see if a vehicle can currently be passed to them. The `spaceAvailableOnOutgoing` functions should be passed to the `isSpaceAvailable` exposures of the junction's outgoing lanes. `handoverVehicleToOtherA`, plus its B and C equivalents, are also imported by `Nto3VehicleManagerComponent`. These functions should be bound to the `handoverVehicle` functions exported by the junction's three outgoing lanes, and allow the junction to inform these lanes that it has passed a vehicle to it. `handoverVehicleToOtherA` also imports a function named `moveVehicle`, which must be bound to the `moveInstance` function exposed by the Fluid framework's System instance. `moveVehicle` takes two parameters: the first is the relative or absolute path of where to move a vehicle instance from, and the second is the path of where to move vehicle to. Finally, `Nto3VehicleManagerComponent` imports the `onUpdate` event as exposed by the `UpdateSimulationComponent` described above.

The configuration of `Nto3VehicleManagerComponent` instances must include the junction's position in two-dimensional Euclidean space so that the junction may be rendered correctly. Meanwhile, the `outgoingInstancePathA` value, along with its B and C versions, provide the absolute or relative paths for outgoing lane instances. Although the paths for outgoing lanes could be more conveniently accessed from the list of imported functions, the Fluid framework restricts access to all but the `Configuration` section of OODDP descriptions during component instantiation, and so the information must be duplicated. `switchLanesAtFrame` instructs the junction to cycle its currently active outgoing lane after a given number of time frames.

Listing 3.4: The component specification for the Nto3VehicleManagerComponent component.

```
<Exports>
  <Function id="handoverVehicle">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
        <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="isSpaceAvailable">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
</Exports>
<Imports>
  <Function id="spaceAvailableOnOutgoingA">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
  <Function id="spaceAvailableOnOutgoingB">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
  <Function id="spaceAvailableOnOutgoingC">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
  <Function id="handoverVehicleToOtherA">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="handoverVehicleToOtherB">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="handoverVehicleToOtherC">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="moveVehicle">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters>
        <Parameter xsi:type="PrimitiveType" typename="String"/>
        <Parameter xsi:type="PrimitiveType" typename="String"/>
      </Parameters>
    </Type>
  </Function>
  <Event id="onUpdate">
    <Type><Payload xsi:type="PrimitiveType" typename="Float"/></Type>
  </Event>
  <Event id="onRender">
    <Type><Payload xsi:type="PrimitiveType" typename="Nil"/></Type>
  </Event>
</Imports>
<Configuration>
  <Value id="position">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
  <Value id="switchLanesAtFrame">
    <Type xsi:type="PrimitiveType" typename="Float"/>
  </Value>
  <Value id="outgoingInstancePathA">
    <Type xsi:type="PrimitiveType" typename="String"/>
  </Value>
  <Value id="outgoingInstancePathB">
    <Type xsi:type="PrimitiveType" typename="String"/>
  </Value>
  <Value id="outgoingInstancePathC">
    <Type xsi:type="PrimitiveType" typename="String"/>
  </Value>
</Configuration>
```

Lane behaviour In a similar way to junctions, each lane is modelled as an OODDP instance with a `VehicleLaneComponent` providing its runtime behaviour. Each lane acts as a container for vehicle instances, with `VehicleLaneComponent` accessing individual vehicles via its concept of self. A lane's response to an `UpdateSimulationComponent` update event is to iterate through the vehicles contained by its enclosing instance, incrementing each vehicle's position. Vehicles are updated in order, starting with the vehicle furthest along the lane and until the rearmost vehicle is processed. The position of each vehicle is incremented according to the current time scale, although the lane must also maintain a given inter-vehicle distance. When a vehicle reaches the end of the lane, `VehicleLaneComponent` passes the vehicle to its outgoing junction.

When considering how lanes are to be connected to incoming and outgoing junction instances, the same push and pull options as described above are available. Lanes may either push vehicle instances to their outgoing junctions, or pull vehicles from their incoming junctions in response to an event. In this case, the lanes in the traffic simulation will use push semantics in order to remain consistent with the behaviour of junctions. The specification document for the `VehicleLaneComponent` given in Listing 3.5 is subsequently similar to that of a junction with one outgoing lane. However, lane components require two additional elements in their configurations, providing the two dimensional position of its incoming and outgoing junctions. These positions are used to correctly render the lane as connecting the two junctions together.

Vehicle behaviour As described above, the vehicles in the traffic simulation do not exhibit any complex behaviour of their own. Consequently, they do not require a component providing runtime behaviour in response to an `UpdateSimulationComponent` update event. Each vehicle object is represented by an OODDP instance, which includes the vehicle's diffuse color as a nested Value in order to vary vehicle appearance during simulation visualisation. Vehicle instances are created at runtime by vehicle sinks, as described below.

Vehicle sources and sinks While the junctions and lanes in the traffic network are collectively responsible for the movement of vehicles, the traffic system also requires a method for creating vehicle instances and removing those leaving the simulated area.

The vehicle source component is responsible for creating vehicle instances. A vehicle source is similar to a junction with no incoming lanes and a single outgoing lane. In response to an `UpdateSimulationComponent` update event, a vehicle source will create a single vehicle instance and pass that instance to its outgoing lane.

The vehicle sink component is responsible for the destruction of vehicle instances and thereby removing them from the traffic simulation. A vehicle source is similar to a junction with one incoming

Listing 3.5: The component specification for the VehicleLaneComponent component.

```
<Exports>
  <Function id="handoverVehicle">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="isSpaceAvailable">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
</Exports>
<Imports>
  <Function id="spaceAvailableOnOutgoing">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters/>
    </Type>
  </Function>
  <Function id="handoverVehicleToOther">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters><Parameter xsi:type="PrimitiveType" typename="String"/></Parameters>
    </Type>
  </Function>
  <Function id="moveVehicle">
    <Type>
      <ReturnType xsi:type="PrimitiveType" typename="Boolean"/>
      <Parameters>
        <Parameter xsi:type="PrimitiveType" typename="String"/>
        <Parameter xsi:type="PrimitiveType" typename="String"/>
      </Parameters>
    </Type>
  </Function>
  <Event id="onUpdate">
    <Type><Payload xsi:type="PrimitiveType" typename="Float"/></Type>
  </Event>
  <Event id="onRender">
    <Type><Payload xsi:type="PrimitiveType" typename="Nil"/></Type>
  </Event>
</Imports>
<Configuration>
  <Value id="incomingInstancePosition">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
  <Value id="outgoingInstancePosition">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
  <Value id="absoluteOutgoingInstancePath">
    <Type xsi:type="PrimitiveType" typename="String"/>
  </Value>
</Configuration>
```

lane and no outgoing lanes. When an incoming lane pushes a vehicle instance onto a vehicle sink, the vehicle sink will immediately destroy the vehicle instance.

Visualisation The OpenGLContextComponent provides support for the traffic simulation's runtime visualisation by creating and maintaining a window and rendering surface that may receive rendering instructions via the OpenGL rendering API. The OpenGLContextComponent also encapsulates essential rendering behaviours, such as placing the viewpoint, allowing the traffic simulation's constituent parts to draw themselves during the current time frame, and clearing the window ready for the next time frame. In order to synchronise the runtime behaviour of the traffic simulation with its visualisation, the OpenGLContextComponent subscribes to the UpdateSimulationComponent update event, and emits a render event each time frame. Visible traffic system elements are required to subscribe to the render event, and respond by calling appropriate OpenGL functions so that their visual representations are drawn.

Listing 3.6 gives the component specification document for the OpenGLContextComponent. Lines 2 and 7 respectively export the OpenGLContextComponent onRender event and import the UpdateSimulationComponent onUpdate event as discussed above. The OpenGLContextComponent also imports the stopApplication function exported by UpdateSimulationComponent in order to allow users to terminate the traffic simulation by closing the rendering window, as shown by Line 12.

Lines 20, 25 and 30 in the OpenGLContextComponent configuration allow the visualisation's viewpoint to be set, with value names and types corresponding to the parameters of the GLUT gluLookAt function[10]. position is the 3D Euclidean position of the virtual camera, while lookAt gives the 3D position of what is being viewed. upVector is a normalised 3D vector representing the upward orientation of the camera[11].

3.3.3 Application Configuration

The Fluid traffic simulation's application configuration consists of over 1,000 lines of XML. Section 3.3.3 will describe the most relevant parts of the application configuration and highlight areas for further discussion. Note that isSpaceAvailable and spaceAvailableOnOutgoing connections have been removed in order to improve the clarity of the following discussion.

Listing 3.7 provides the traffic simulation's OODDP Lane type definition. The imports specified as part of Listing 3.7 must match those of the VehicleLaneComponent component specification, as given in Listing 3.5. The Lane type connects the Fluid System instance's moveInstance

[10]The Graphics Library Utility Toolkit (GLUT) provides a number of extensions to the functionality provided by the OpenGL API, and is often distributed alongside the OpenGL library.
[11]The 'up' vector is typically set to { 0, 1, 0 } in order to coincide with the positive Y axis, which is commonly associated with the vertical direction in 3D graphics.

Listing 3.6: The component specification for the OpenGLContextComponent component.

```
<Exports>
  <Event id="onRender">
    <Type><Payload xsi:type="PrimitiveType" typename="Nil"/></Type>
  </Event>
</Exports>
<Imports>
  <Event id="onUpdate">
  <Type>
    <Payload xsi:type="PrimitiveType" typename="Nil"/>
  </Type>
  </Event>
  <Function id="stopApplication">
    <Type xsi:type ="FunctionTypeNode">
      <ReturnType xsi:type="PrimitiveType" typename="Nil"/>
      <Parameters/>
    </Type>
  </Function>
</Imports>
<Configuration>
  <Value id="position">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
  <Value id="lookAt">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
  <Value id="upVector">
    <Type xsi:type="SequenceType">
      <Type xsi:type="PrimitiveType" typename="Float"/>
    </Type>
  </Value>
</Configuration>
```

function to the `VehicleLaneComponent moveVehicle` import. The `UpdateSimulationComponent onUpdate` event and the `OpenGLContextComponent onRender` event are also imported, so that the `VehicleLaneComponent` will be informed of time frame updates and when to render its visual representation.

As the `Lane` type is the base OODDP type for all lane instances used to define the traffic network, lane instances will not have to form these basic connections individually, but will instead inherit them from the base type. Each lane instance will provide the position of its incoming and outgoing network nodes for visualisation purposes. Lane instances are also required to define their connectivity to vehicle sources, vehicle sinks and junctions as discussed below.

Listing 3.8 gives the OODDP type description for the `3Way` type, which provides the base class for all junctions with N incoming lanes and 3 outgoing lanes. The type description in Listing 3.8 must match the specification given in Listing 3.4. The `3Way` type includes the same subscriptions to `moveInstance`, `onUpdate` and `onRender` as described for the `Lane` type. However, the `3Way` type is also able to stipulate the paths for its `handoverVehicleToOther` functions and `outgoingInstancePath` configurations for lanes A, B, and C. These topological references are allowed here because of the way in which junction and lane instances are arranged. As shown in Listing 3.9 and Figure 3.10b, each junction instance contains the definition of its outgoing lanes as nested OODDP instances. The relationship between junctions and their outgoing lanes is consequently fixed, and the relationships may be described as part of the base OODDP types for 1, 2 and 3 way junctions.

Listing 3.9 provides the OODDP instance description for node `b` in the traffic network. As node `b` is a junction with 3 outgoing lanes, its instance description includes the `3Way` type as its `parent`, and will subsequently inherit, override and extend the description given in Listing 3.8.

Despite inheriting a number of XML elements from the `Lane` and `3Way` parent types, the OODDP instance description given in Listing 3.9 is still unnecessarily verbose. The length of the OODDP instance description for node `b` is mainly due to the need to provide both type information and data for values. In particular, each 2D position given for lanes and junctions requires 10 lines of XML. The expressiveness of Fluid's composition language could be improved by allowing regular patterns to be defined and reused. For example, a `2DPosition` could be defined as the sequence of two floating point numbers, and the 2D position for node `b` could subsequently be given as follows:

```
<2DPosition>
  <Data>
    <Float>5.0</Float>
    <Float>0.0</Float>
  </Data>
</2DPosition>
```

OODDP type and instance descriptions could be further improved by allowing component instances to access the entirety of such definitions during their instantiation. This change to Fluid's component model would allow lane instances to reuse the outgoing junction path given in the `handoverVehicleToOther` function import, avoiding the need to provide the path again as part of the lane's configuration.

3.3.4 Fluid Application Execution

Figure 3.11 shows the runtime operations of the Fluid executable when running the traffic simulation example. The illustration given by Figure 3.11 is based on the description of the Fluid executable

Listing 3.7: The OODDP description for a Lane type in the traffic simulation.

```
<Type id="Lane">
  <Component id="vehicleManager" type="VehicleLaneComponent">
    <Imports>
      <Event id="onUpdate"
        exportedEvent="root/SimulationControl/updater/onUpdate"/>
      <Event id="onRender"
        exportedEvent="root/SimulationControl/OpenGLRenderer/onRender"/>
      <Function id="moveVehicle"
        exportedFunction="root/system/namespaceManagement/moveInstance"/>
    </Imports>
    <Configuration/>
  </Component>
</Type>
```

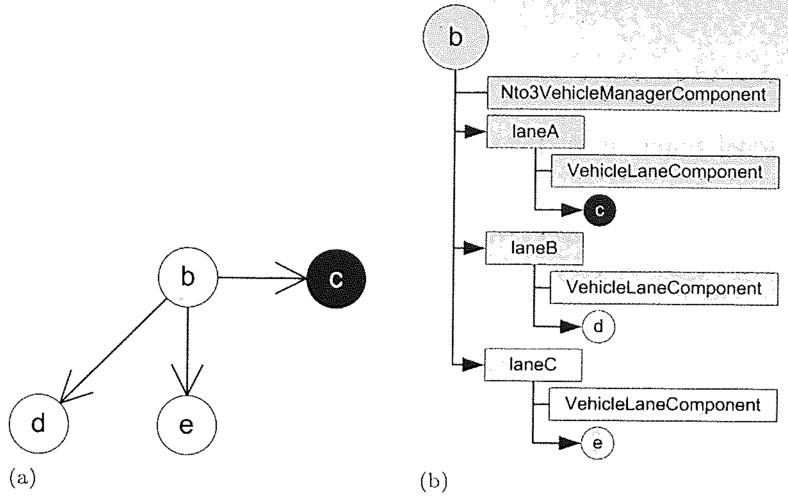


Figure 3.10: Two views of the 3 way junction at node b from Figure 3.9. (a) shows the junction's topology from Figure 3.9, while (b) illustrates the runtime structure of the OODDP instances. Lines with arrows in (b) show the direction of `handoverVehicleToOther` function connections.

Listing 3.8: The OODDP description for a Junction type with N incoming lanes and 3 outgoing lanes.

```
<Type id="3Way">
  <Component id="vehicleManager" type="Nto3VehicleManagerComponent">
    <Imports>
      <Event id="onUpdate"
        exportedEvent="root/SimulationControl/updater/onUpdate"/>
      <Event id="onRender"
        exportedEvent="root/SimulationControl/OpenGLRenderer/onRender"/>
      <Function id="moveVehicle"
        exportedFunction="root/system/namespaceManagement/moveInstance"/>
      <Function id="handoverVehicleToOtherA"
        exportedFunction="laneA/vehicleManager/handoverVehicle"/>
      <Function id="handoverVehicleToOtherB"
        exportedFunction="laneB/vehicleManager/handoverVehicle"/>
      <Function id="handoverVehicleToOtherC"
        exportedFunction="laneC/vehicleManager/handoverVehicle"/>
    </Imports>
    <Configuration>
      <Value id="outgoingInstancePathA" xsi:type="PrimitiveValue">
        <Type typename="String"/><Data><String>laneA</String></Data>
      </Value>
      <Value id="outgoingInstancePathB" xsi:type="PrimitiveValue">
        <Type typename="String"/><Data><String>laneB</String></Data>
      </Value>
      <Value id="outgoingInstancePathC" xsi:type="PrimitiveValue">
        <Type typename="String"/><Data><String>laneC</String></Data>
      </Value>
    </Configuration>
  </Component>
</Type>
```

Listing 3.9: The OODDP description for a Junction instance with N incoming lanes and 3 outgoing lanes. Some lines have been removed for clarity.

```
<Instance id="b" parent="3Way">
  <Component id="vehicleManager" type="Nto3VehicleManagerComponent">
    <Imports/>
    <Configuration>
      <Value id="position" xsi:type="SequenceValue">
        <Type>
          <Type xsi:type="PrimitiveType" typename="Float"/>
          <Type xsi:type="PrimitiveType" typename="Float"/>
        </Type>
        <Data>
          <Datum xsi:type="PrimitiveData"><Float>5.0</Float></Datum>
          <Datum xsi:type="PrimitiveData"><Float>0.0</Float></Datum>
        </Data>
      </Value>
    </Configuration>
  </Component>
  <Instance id="laneA" parent="root/trafficSim/Lane">
    <Component id="vehicleManager"  type="VehicleLaneComponent">
      <Imports>
        <Function id="handoverVehicleToOther"
          exportedFunction="root/trafficSim/c/consumer/handoverVehicle"/>
      </Imports>
      <Configuration>
        <Value id="incomingInstancePosition" xsi:type="SequenceValue">
          <Type>
            <Type xsi:type="PrimitiveType" typename="Float"/>
            <Type xsi:type="PrimitiveType" typename="Float"/>
          </Type>
          <Data>
            <Datum xsi:type="PrimitiveData"><Float>5.0</Float></Datum>
            <Datum xsi:type="PrimitiveData"><Float>0.0</Float></Datum>
          </Data>
        </Value>
        <Value id="outgoingInstancePosition" xsi:type="SequenceValue">
          <Type>
            <Type xsi:type="PrimitiveType" typename="Float"/>
            <Type xsi:type="PrimitiveType" typename="Float"/>
          </Type>
          <Data>
            <Datum xsi:type="PrimitiveData"><Float>10.0</Float></Datum>
            <Datum xsi:type="PrimitiveData"><Float> 0.0</Float></Datum>
          </Data>
        </Value>
        <Value id="absoluteOutgoingInstancePath" xsi:type="PrimitiveValue">
          <Type typename="String"/>
          <Data><String>root/trafficSim/c</String></Data>
        </Value>
      </Configuration>
    </Component>
  </Instance>
  <Instance id="laneB" parent="root/trafficSim/Lane">
    <!-- Contents are similar to those for laneA (removed for clarity) -->
  </Instance>
  <Instance id="laneC" parent="root/trafficSim/Lane">
    <!-- Contents are similar to those for laneA (removed for clarity) -->
  </Instance>
</Instance>
```

given in Section 3.2.4, but with concrete references to elements of the traffic system example described above.

3.3.5 Results

When executed, the traffic simulation described above operates at interactive frame rates. Figures 3.12a, 3.12b and 3.12c show the visualisation produced by time frames 500, 1000 and 1500 with a frame delta of 0.1 during a single execution of the traffic simulation. Note that the traffic network becomes congested due to the high density of traffic.

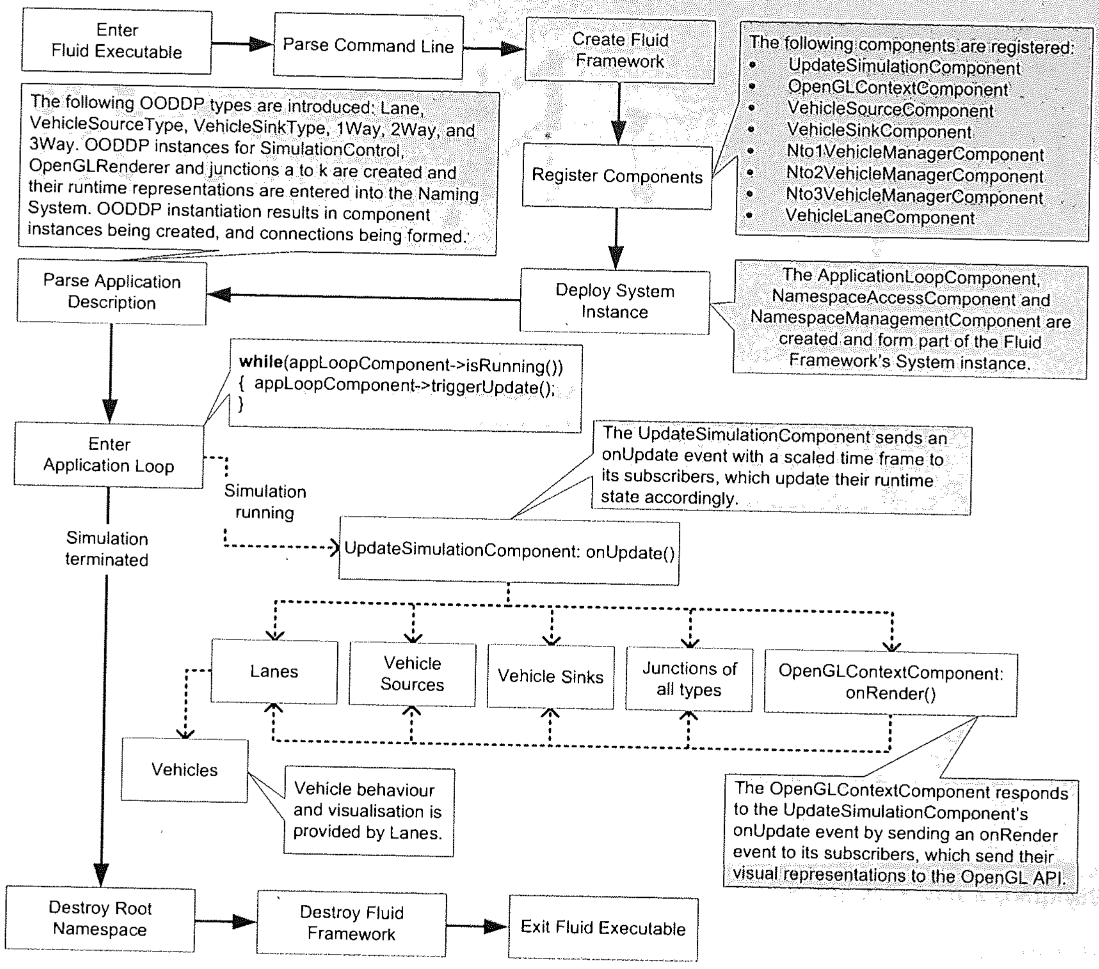
The traffic network shown in Figures 3.13a, 3.13b and 3.13 has a higher traffic light switching frequency for both 2Way and 3Way junctions. The traffic is subsequently more dispersed, and vehicles are able to find their way to vehicle sinks without the congestion seen in Figure 3.12.

3.3.6 Critical Evaluation

Section 3.3 describes the successful application of the Fluid framework to model a complete albeit simple traffic simulation. As shown in Section 3.3.5, the Fluid application satisfies the behaviour described in Section 3.3.1. However, as discussed throughout sections 3.3.2, and 3.3.3, the development of the traffic simulation example has not been trivial, and a number of limitations of the Fluid framework have been highlighted.

The difficulties encountered when designing and implementing the traffic simulation are partly due to restrictions and limitations of the Fluid framework itself. Certain aspects of Fluid's component model improve system robustness by reducing the flexibility available to component implementations. However, such restrictions have further reaching effects on Fluid's composition language and the OODDP concepts it supports. Furthermore, the design of abstractions supporting component topologies could be improved to enhance the expressiveness of Fluid application configurations. An event system based on message broadcast, rather than explicitly subscribing to the event publisher, would also make inter-component connections more flexible to change. Finally, the use of a declarative XML based composition language has resulted in component specifications and application configurations that are overly verbose and unclear. If the Fluid framework is to continue making use of XML based technologies, then there is an obvious requirement for user friendly editing tools or higher level abstractions supported by Fluid's composition language.

Section 3.3.3 has demonstrated the use of OODDP concepts and inheritance manipulations in order to improve the expressiveness of XML based application configurations. Listings 3.8, 3.7 and 3.9 together illustrate how OODDP can support an object oriented type system making use of inheritance relationships to reuse fundamental definitions in derived instantiations. While the application config-



Enter Fluid Executable
Parse Command Line
Create Fluid Framework
The following components are registered:
• UpdateSimulationComponent
• OpenGLContextComponent
• VehicleSourceComponent
• VehicleSinkComponent
• Nto1VehicleManagerComponent
• Nto2VehicleManagerComponent
• Nto3VehicleManagerComponent
• VehicleLaneComponent
The following OODDP types are introduced: Lane, VehicleSourceType, VehicleSinkType, 1Way, 2Way, and 3Way. OODDP instances for SimulationControl, OpenGLRenderer and junctions a to k are created and their runtime representations are entered into the Naming System. OODDP instantiation results in component instances being created, and connections being formed.
Register Components
Parse Application Description
Deploy System Instance
The ApplicationLoopComponent, NamespaceAccessComponent and NamespaceManagementComponent are created and form part of the Fluid Framework's System instance.
while(appLoopComponent->isRunning()) { appLoopComponent->triggerUpdate(); }
Enter Application Loop
Simulation running
The UpdateSimulationComponent sends an onUpdate event with a scaled time frame to its subscribers, which update their runtime state accordingly.
Simulation terminated
UpdateSimulationComponent: onUpdate()
Lanes
Vehicle Sources
Vehicle Sinks
Junctions of all types
OpenGLContextComponent: onRender()
Vehicles
Vehicle behaviour and visualisation is provided by Lanes.
The OpenGLContextComponent responds to the UpdateSimulationComponent's onUpdate event by sending an onRender event to its subscribers, which send their visual representations to the OpenGL API.
Destroy Root Namespace
Destroy Fluid Framework
Exit Fluid Executable

Figure 3.11: An overview of the Fluid component executable's runtime operation in the context of the traffic simulation as described in Section 3.3. Solid lines with closed arrows show the high level control flow of the Fluid executable, while dotted lines with open arrows indicate event propagation.

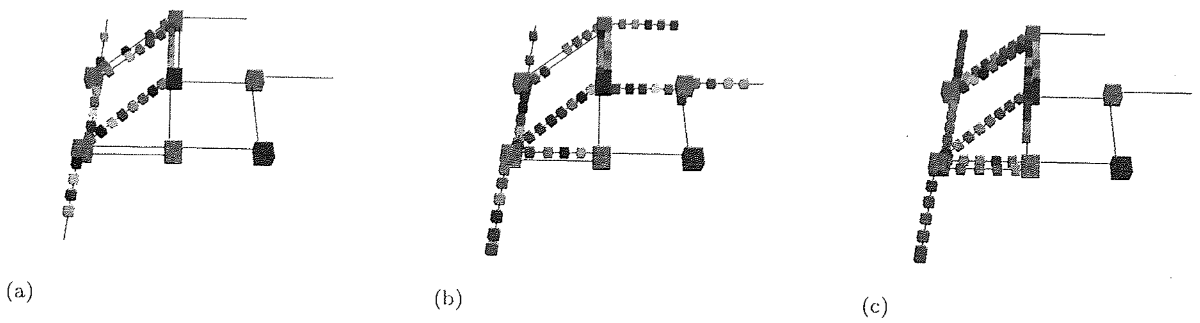


Figure 3.12: Screen shots taken at time frames 500, 1000 and 1500 of an execution of the traffic simulation, showing network gridlock.

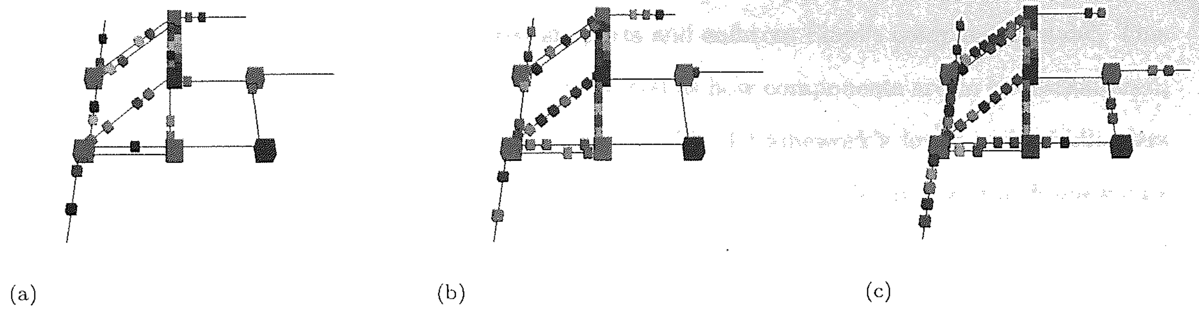

(a) (b) (c)

Figure 3.13: Screen shots taken at time frames 500, 1000 and 1500 of an execution of the traffic simulation after making a small number of changes to the application configuration.

uration file for the traffic simulation example may be overly long and difficult to understand, OODDP has both reduced its length and improved the expressiveness of the concepts described. Moreover, the availability of OODDP concepts has helped to manage the complexity of the traffic simulation's design and implementation, by allowing reusable concepts to be encapsulated, named and inherited as part of a high level composition language. An application configuration based on OODDP abstractions embodies a number of structures at varying scales of granularity, which may be comprehended and manipulated in a similar way to OO classes.

The successful application of OODDP in the traffic simulation example suggests that a composition language incorporating additional levels of abstraction may be more powerful and expressive than that supported by the Fluid framework's current design and implementation. One goal would be to approach the expressiveness and capabilities of modern integration languages such as Lua. A Lua prototype of the traffic simulation producing the behaviour described in Section 3.3.1 was developed before writing the Fluid implementation described in Sections 3.3.2 and 3.3.3. The Lua prototype was easier to write, with a more flexible and capable implementation, and required fewer than 500 lines of code[12].

3.4 Summary

This chapter has described the Fluid component framework, including the aims and motivations leading to its development in Section 3.1, followed by relevant details of its design and implementation in Section 3.2. Section 3.2.4 provides a complete overview of the Fluid framework's runtime behaviour, while Section 3.3 gives an example application.

The Fluid framework combines mainstream CBSD technologies with OODDP from the game development field. The framework's implementation incorporates a low level platform augmenting the

[12]The Lua prototype makes use of a third party library to provide OpenGL support.

features of the C++ language in order to provide a range of low level functionality including additional type information, as well as standardised data representations, manipulation, and storage. Building upon its lower tier, the framework's middle tier supports and enforces Fluid's component model. This model defines what constitutes a Fluid component, as well as how components are to be instantiated, parameterised, communicated with, and destroyed. The Fluid framework's lower and middle tiers thus constitute a component based approach for the C++ platform. Meanwhile, the framework's higher tier provides a range of high level abstractions for manipulating compositions, while focussing on controlling component parameterisation via a number of OODDP concepts and operations.

The resulting development environment is consequently minimal, with component implementation and specifications responsible for the definition of application functionality, appropriate interfaces, and the topology of component deployments. The Fluid framework does not directly support the definition of geovisualisation or other spatiotemporal applications, as its implementation does not provide interfaces or other abstractions related to a particular application domain. Instead, the Fluid framework provides a fixed range of behaviours which collectively allow application descriptions to drive the population of the naming system, the deployment and parameterisation of components, and the propagation of function calls and events. The framework does not dictate the granularity, delimitation or interface of application functionality, but is instead responsible for supporting the description, instantiation, parameterisation and runtime operation of a wide range of Fluid applications.

The Fluid framework represents a vast improvement over its predecessor in terms of flexibility and extensibility. However, the framework's design does not remove all of the prototype's limitations, and aspects of its implementation may benefit from further improvement.

The Fluid framework retains reliance on implicit rules to define concepts provided by the combination of two or more components, although the framework is able to solve some cases using OODDP representations. For example, in the Fluid prototype a visible scene object is defined by an OODDP type or instance containing both geometry and appearance facet objects. In the Fluid framework, a renderable could be represented by a `Renderable` OODDP type with an appropriate interface, with derived types and instances providing custom implementations for the separate geometry and appearance concepts. During the application's execution, runtime type information could be used to select those OODDP instances deriving from the `Renderable` type, which could then be passed to an appropriate component for rendering.

However, a Fluid application must still make use of an implied relationship in order to communicate geometry to the render system, as Fluid's standardisation of visible types requires that matrix and vertex objects must be translated to Value manipulations. This problem may be solved by allowing component specifications to define their own names for Value manipulations, and to introduce them to the framework's type system. For example, a `Matrix4x4` type, described by an appropriate render system component, could also be used by the `Renderable` type described above in order to explicitly model an OpenGL matrix concept. The introduction of aliases would also facilitate more consistent type checking between component implementations, and allow for a number of widely accepted data representations to be built upon those natively supported by the Fluid framework.

Further limitations are introduced as a result of the rapid design and development of the Fluid framework; indeed, the Fluid framework's current incarnation may be considered a prototype or proof of concept focussing on the combination of CBSD and OODDP technologies. Consequently, much of the framework's implementation provides the required functionality, but may not represent the optimal solution. Furthermore, the framework's design does not consider problems such as the distributed deployment of components, nor their runtime operation over multiple threads of execution.

A number of errors may also exist in the framework's current implementation, limiting its use in practice. For example, the requirement for runtime manipulation and adaptation of component topologies was introduced relatively late in the development process, and was not considered during the initial design of the framework's lower level tiers. Consequently, function and event bindings do not migrate consistently when moving OODDP instances from one location in the naming system to another. It is expected that such limitations can be removed during the subsequent maintenance, and improvement of the framework's implementation as the Fluid project matures.

Finally, a number of limitations of the Fluid framework are due to the C++ platform. As discussed in Section 3.2.1, the development of the framework's lower tiers involved introducing a range of functionality that already exists in more appropriate languages and platforms. For example, the .NET framework supports a number of powerful development languages such as C♯, includes support for a deployment concept in the form of assemblies, and is particularly suited to CBSD. Modern languages such as Java and C♯ are further aided by features such as reflection, code generation, and the use of attributes, which allow the use of attribute oriented programming [49]. It is speculated that a number of improvements may be made by adopting a higher level programming language and development platform in future developments of the Fluid framework.

The initial inception and early design of the Fluid framework highlighted a number of research questions regarding the validity, applicability, flexibility and expressiveness of a CBSD technology incorporating an OODDP approach. As described throughout Sections 3.2.2 and 3.2.3, and further demonstrated in Section 3.3, OODDP concepts and their manipulations have been successfully combined with CBSD in the Fluid framework.

The resulting system maintains the flexibility and expressiveness of OODDP, supporting a range of high level concepts with which to define and manipulate a number of highly configurable, small scale components with data driven behaviours. However, the Fluid framework also broadens the application of data driven programming, to include the selection and topology of computational encapsulations. The Fluid project thus incorporates the principles of CBSD, allowing applications to be developed using independently developed binary components that may be deployed, interconnected and parameterised in order to form a variety of Fluid applications.

Chapter 4

Contextualising the Fluid Framework

The evolution of the Fluid framework, from its prototypical OO framework to its current incarnation as a component framework and composition language, is the result of a multidisciplinary research project spanning several fields as described in Chapter 2. Figure 4.1, taken from Chapter 2, illustrates these contributing fields.

This chapter places the work carried out in this research within the context of these surrounding fields; Sections 4.1, 4.2, and 4.3 highlight the contributions made to the role of data in applications, to software development methodologies, and to game development technologies respectively.

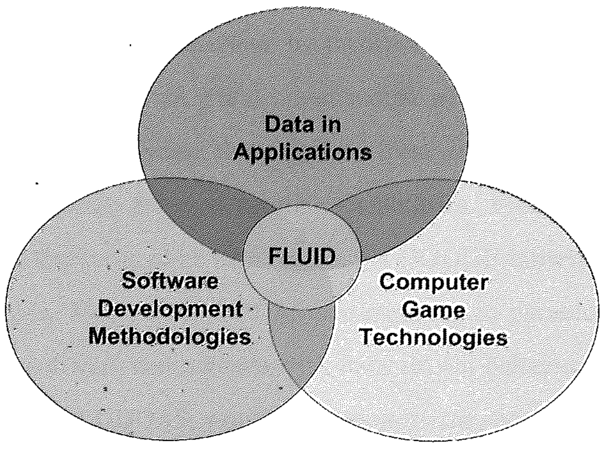

Data in Applications
FLUID
Software Development Methodologies
Computer Game Technologies

Figure 4.1: A Venn diagram including the domains contributing to the project.

4.1 Data in Applications

The evolving relationship between application data and software has reached a point where empowered data formats are able to drive, determine, or significantly influence the runtime behaviour of a wide range of application functionality. The resulting *data driven paradigm* is the focus of DDDAS, where measurement data and software behaviour form a synergistic cycle, as illustrated by Figure 4.2a. The Fluid project does not aim to create a complete GIS or DDDAS solution: GIS and DDDAS are neither small nor simple software applications, and their successful development typically involves the multidisciplinary collaboration of numerous individuals.

Instead, the Fluid project embodies the amalgamation of a range of technologies including GIS, DDDAS, OODDP and CBSD, in order to form a novel type of data driven manipulation and functionality. Although it has not been possible to integrate Fluid with any new or existing DDDAS platforms, one can speculate the role of the Fluid framework as part of a Fluid-based DDDAS application. As illustrated by Figure 4.2b, a Fluid-based DDDAS would use measurement data to drive the creation or adaptation of one or more Fluid applications, which in turn provide the DDDAS simulation.

The approach described here lies outside those techniques currently used to support dynamic behaviour in DDDAS, which typically adjust their models by modifying parameter values according to incoming data. Although the resulting systems are able to exhibit a range of data driven behaviour, the *capabilities* of current DDDAS software remain static: parameterisation cannot introduce new functionality, but is limited to changing the behaviour of existing code by directing control flow via the manipulation of branch conditions or loop counters.

By contrast, a Fluid-based DDDAS could potentially facilitate the dynamic data driven modification of application capabilities and behaviour by incorporating a variety of OODDP types and instances as part of its runtime model. A Fluid-based model would be open to the introduction of new OODDP types and instances, including those derived from existing OODDP types, forming new software structures with dynamically selected runtime capabilities. Fluid's composition language allows new OODDP types and instances to be introduced, via OODDP inheritance, by describing their specialisation of existing types. Meanwhile, DDDAS can continue to parameterise existing functionality due to Fluid's focus on software components with data driven behaviour. A Fluid-based model, operating as part of a Fluid-based DDDAS, would thus support the dynamic modification of software behaviour and capabilities at various scales of abstraction and granularity, by allowing configuration data to drive the selection, interconnection and configuration of Fluid components.

A Fluid approach may not be suitable to all application domains of DDDAS; indeed, there are areas where a Fluid-based DDDAS may be particularly effective, where DDDAS may gain partial benefit from incorporating Fluid technologies, and where a Fluid approach may not be appropriate

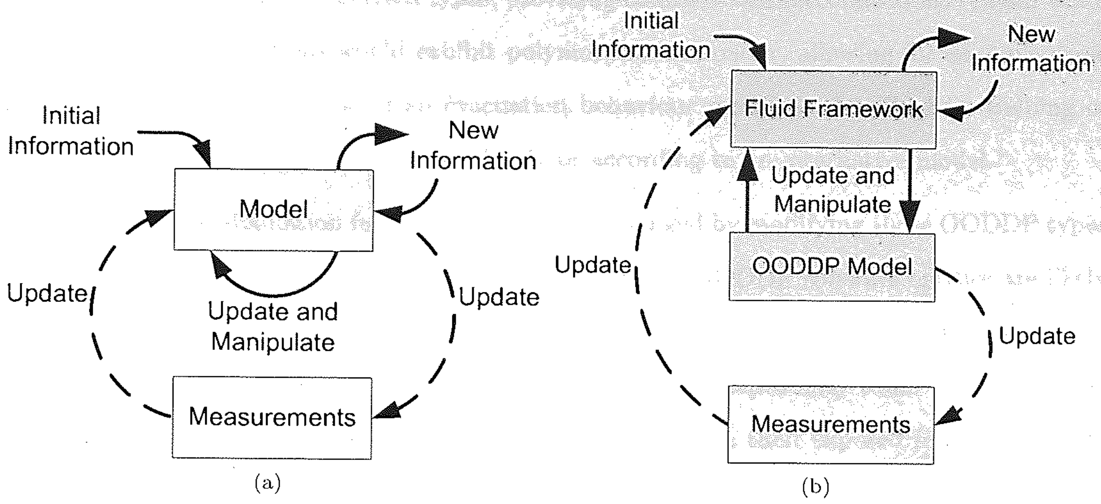


Figure 4.2: An illustration of the functionality available to DDDAS (a) and a speculated Fluid-based DDDAS (b). Dashed lines in both (a) and (b) denote the synergistic feedback loop formed by DDDAS measurement and simulation.

at all. This scale of applicability is described below with the use of some illustrative speculations:

Areas most suited to a Fluid-based DDDAS The Fluid approach described above has greatest applicability to DDDAS focusing on discrete phenomena with varying individual attributes and dynamic behaviour. Relevant examples include the evacuation of a nightclub fire as studied by Chaturvedi et al. [19], the management of surface transportation systems in the work of Fujimoto et al. [21], and informing emergency response systems using wireless phone signals as seen in Madey et al. [133].

DDDAS in this category offer opportunities to incorporate Fluid's OODDP software descriptions at two levels: the first (micro) level consists of the description of the various elements constituting the simulation, including elements such as nightclub patrons, vehicles and cell phone users; meanwhile, the second (macro) level consists of the simulation software itself, including its various responsibilities for receiving input, updating the simulation, and providing any required outputs, which may include realtime visualisations.

At the micro level, Fluid's OODDP descriptions may be used to provide a range of individually parameterised entities that collectively form a given simulation. For example, a wide variety of nightclub patrons could be described and instantiated by first describing a default or base 'person' type and then providing individual characteristics as part of OODDP inheritance or instantiation manipulations. Deeper OODDP inheritance hierarchies could be used to describe vehicle types with distinct appearances and behaviours, leading to simulations populated with a variety objects having closer correspondence to their real-world counterparts.

Meanwhile, Fluid's component model allows components of base types to be exchanged with overriding behaviour in their derived types, providing component interconnections remain valid. The resulting configurations would exhibit polymorphic behaviour, allowing for example some nightclub patrons to make use of an evacuation behaviour model as described by Helbing et al. [134], while other patrons act independently or according to an alternative model.

At the macro level, simulation functionality could be changed by modifying those OODDP types and instances used to form the simulation software. These OODDP software entities are likely to be larger in scale than those used to populate the simulation, and will typically operate at a higher level of abstraction. The current level of DDDAS adaptability could be achieved by modifying the parameterisation of such types and instances via their exposed functions, events and Value instances. However, a greater degree of change could be supported by modifying existing OODDP instances or using OODDP inheritance manipulations to derive new instances providing additional functionality. For example, a Fluid-based DDDAS could introduce OODDP types providing support for additional data input, user interaction, new simulation management strategies, or more realistic visualisation methods. Software behaviour introduced in this way could be added to existing simulations through the instantiation of OODDP types, appropriate event listeners and function bindings.

DDDAS focusing on discrete phenomena with varying individual attributes and dynamic behaviour are most appropriate targets for Fluid integration; indeed, there is a clear opportunity for the use of OODDP types, instances and manipulations at varying levels of abstraction. Furthermore, the use of OODDP inheritance appears most attractive where a range of related scenarios are to be simulated (for example, variations of a nightclub's architecture [19]). The Fluid approach to this problem would be to describe each scenario as an OODDP instance deriving from a common default case. Fluid's support for OODDP inheritance manipulations would allow each scenario to be described as nothing more than its variations from the default template.

Areas partially suited to a Fluid-based DDDAS Many domains already benefit from dedicated software solutions that are widely used, trusted, and accepted as standard. For example, areas such as weather analysis and forecasting in the work of Plale et al. [135] and the analysis of urban water distribution systems as studied by Mahinthakumar et al. [136] make use of a range of software applications, middleware, and hardware devices, that form part of well established solutions. These studies are unlikely to benefit from replacing their current solutions with a novel approach based on the Fluid framework, as their micro-scale elements are often entirely managed by dedicated libraries providing a range of specialised functionality.

However, one can speculate potential benefit from adopting a Fluid approach at the macro scale, where existing solutions appear to be seeking additional support [136]. While domain-specific solutions provide specialised low level behaviour in order to support data modelling, simulation and analysis, a higher level Fluid approach could determine overall application control or work flow. For example, a Fluid-based DDDAS would allow widespread changes to be made to application parameterisation using OODDP inheritance manipulations. Furthermore, existing OODDP instances could be exchanged with appropriately derived objects in order to modify overall application functionality via polymorphism. The resulting DDDAS workflow would be adaptable to changing requirements, for example including additional data processing or validation, incorporating additional measurements in order to determine measurement reliability.

Areas not particularly suited to a Fluid-based DDDAS While looking forward to the potential applications of Fluid within DDDAS, it is also important to acknowledge those areas where a Fluid-based approach may not be suitable. For example, DDDAS modelling continuous phenomena such as wildfires in the work of Douglas at al. [20] are not well suited to the OO paradigm used by Fluid's composition language, and often make use of dedicated mathematical libraries to provide their lower level functionality. In certain cases, such as the management of water contaminants as studied by Parashar et al. [137], the granularity of adaptation is unclear, and the application of Fluid's OODDP manipulations may be less straightforward. Furthermore, overall application control flow may be necessarily simplistic, or too integrated into lower level details to benefit from the application of Fluid manipulations at the macro scale. In such cases, black box computational encapsulations provide the vast majority of application functionality, and these are often tightly coupled with other software elements. Consequently, existing methods based on adjusting the parameterisation of mathematical models may be more appropriate than a Fluid approach.

The Fluid framework contributes to the evolving role and capabilities of data in applications by incorporating a high level declarative composition language making use of OODDP concepts and manipulations. The Fluid project thus further empowers data by broadening the application of data driven programming to CBSD. The Fluid framework may be used to define a variety of applications as the composition of small scale components with data driven behaviour. The framework has potential applications in many areas where data may be used to drive the deployment of computational encapsulations, the structure of their instantiations, and the behaviour of their runtime execution. Section 4.1 has illustrated potential use of the Fluid framework by speculating a number of applications in the context of DDDAS.

4.2 Software Development Methodologies

The work described here is related to a number of software development technologies and methodologies, many of which have inspired the development of the Fluid prototype and subsequently its evolution to form the Fluid framework. Sections 4.2.1 through 4.2.3 place the Fluid framework and its prototype within the context of surrounding software development methodologies and technologies, progressing from lower level abstractions through to higher level concepts. Section 4.2.1 begins with an overview of how implementation languages, integration languages and platforms relate to this work, while Section 4.2.2 describes links with other component-based technologies including composition languages and component frameworks. Finally, Section 4.2.3 distinguishes the Fluid project from those concepts supported by ADLs, as well as considering the relevant features of certain software development methodologies.

4.2.1 Implementation, integration and platforms

Many of the concepts forming the Fluid framework and its prototype can be found elsewhere in other software development platforms. In a similar way, Fluid's composition language shares certain aspects with several implementation and integration languages. Indeed, the Fluid project as a whole has benefitted from the influence of many implementation languages, integration languages and development platforms in use today.

Popular implementation languages such as C♯ and Java already support component oriented software development, with language features such as standardised data representations, rich runtime type information, reflection, introspection, and a unified concept for component deployment. Furthermore, these features are built upon by platforms such as the .NET framework and JavaBeans, which advocate a component based software development methodology. Such platforms typically provide extensive support for the development of component software, often in the form of a wide range of standard libraries and software components.

The Fluid framework evolved during the course of the project from an initial prototype written in C++. In order to overcome some of the limitations of C++, certain aspects of the technology found in more modern implementation languages have been reproduced in the framework's lower tier. For example, Fluid's type system builds upon the type information exposed by C++ in order to support a range of operations that both C♯ and Java provide as part of the standard language. Furthermore, while Fluid's lower tier clearly reproduces the features provided by certain implementation languages, the component model supported by its middle tier partially resembles the functionality supported by higher level platforms. For example, both the .NET and JavaBeans platforms deliver a range of operations for managing component lifetime and interconnection.

Retrospectively, migrating to the C♯ language and the .NET framework before developing the Fluid framework would have facilitated a more efficient development schedule and more capable implementation. Moreover, the .NET framework would have allowed certain parts of the Fluid prototype to be reused as *unmanaged code*. A Fluid framework written in C♯ would have had the full support of a range of low level functionality, as well as a number of high level abstractions and standard operations for developing, deploying, inspecting and managing components.

Migrating to a more suitable implementation language and supporting software platform would have removed the need to develop certain parts of the Fluid framework's functionality. However, a number of the concepts supported by the framework's lower, middle and higher tiers are more abstract than those typically seen in implementation languages and OO frameworks, and have a closer resemblance to the features of integration languages.

Fluid's higher level abstractions have been strongly influenced by the Lua integration language. For example, the namespace concept used to represent Fluid's deployment space, and supported by the framework's lower tier, closely resembles the table construct that forms the basis of all Lua data structures. In addition, the dynamic typing of Fluid's Values is similarly based on the dynamically typed variables that are characteristic of high level interpreted languages. Furthermore, the OODDP inheritance mechanism supported by Fluid's composition language is very similar to that found in Lua, and many other integration languages providing pseudo-support for OO inheritance through the use of prototypes[1].

However, three main differences separate Fluid's high level concepts from those found in integration languages:

- The first difference is one of abstraction and granularity. Like Fluid, integration languages are typically intended to incorporate concepts ranging from primitive data types to entire subsystem encapsulations. However, the development environment provided by the Fluid framework is much more coarsely grained than that supported by integration languages. Integration languages typically support a continuous range of imperative features with which to define application functionality. These abstractions include variable and constant declarations, expressions and statements, functions and procedures, objects, and other concepts such as Lua's tables. Meanwhile, Fluid offers a fixed enumeration of declarative concepts, namely values and their manipulations, functions and events, components, and its OODDP abstractions. Unlike the continuous descriptions supported by integration languages, the functionality of Fluid applications is delivered by its black-box components.

[1]As opposed to integration languages with native OO support, such as Python.

- Fluid's higher level concepts are further separated from integration languages such as Lua and Python by the degree of flexibility they offer. Integration languages typically provide environments where flexibility is emphasised. To this end, integration languages are commonly imperative programming languages that are both interpreted and dynamically typed. Furthermore, such languages often incorporate high level encapsulations for data structures and algorithms, higher order programming, and tend to provide abstractions for more complex concepts such as multithreading, persistence and implementation deployment.

  In contrast, Fluid gives relatively little control to individual computational elements. Rather than focussing on flexibility, the Fluid framework must balance the flexibility of its higher level representations, including its composition language, against the safety of Fluid applications as a whole. For example, Fluid's hierarchical naming system is partially based on Lua's tables, which remain both accessible and mutable to all parts of a given Lua script throughout its execution. A Fluid application offering the same degree of flexibility would be open to modification by its constituent components, and if a change were made, then the application's runtime structure may no longer reflect that described by the application's initial description. A malicious component could thereby introduce incorrect, inconsistent and potentially dangerous behaviour to a Fluid application.

  In order to provide additional security for Fluid applications, the Fluid framework validates each component's interactions with the framework against both its individual specification and the current application description. Inconsistent interactions, such as attempting to form an inter-component connection that does not appear in the component specification or application description, will violate the component's instantiation and thus prevent the Fluid application from being formed. Furthermore, component implementations have limited access to, and control over, the Fluid framework's naming system and its contents. The framework's System component provides immutable access to the naming system's root node, as well as a number of functions for creating, moving and removing namespaces, Values, OODDP types and OODDP instances. The System component is responsible for ensuring that such modifications are valid, and that the Fluid application remains in a consistent state once the requested changes are made.

- The final difference between Fluid's higher level concepts and integration languages is in the fundamental focus of their use. Integration languages are used to write scripts, typically making use of an imperative or functional notation. Such scripts describe software behaviour as sequences of statements, and are therefore synonymous with software programs. Although some scripts may be used to initialise a software application, and may have some resemblance to application data, such initialisations are applied by executing the script. Initialisation is therefore a side effect of the runtime behaviour described by the script.

On the other hand, Fluid's composition language describes the naming, topology and parameterisation of computational encapsulations using a declarative notation. Furthermore, the inheritance manipulations supported by Fluid's OODDP implementation affect both the behavioural and parameterisation aspects of the types and instances on which they act, but are ultimately concerned with the manipulation of an application description's configuration data.

4.2.2 Components

While certain aspects of the Fluid project can be related to lower level implementation and integration technologies as described in Section 4.2.1, the project as a whole is more directly related to component frameworks and composition languages. This section describes the relationship between the Fluid framework and composition language, and a selection of the most relevant component technologies as described in Section 2.2.3.

CoML The overall design of CoML was influential during the development of Fluid's composition language. CoML is similarly XML based, with a declarative syntax and supporting a connection-oriented component model. In addition, the major elements of CoML are components, properties, functions and events, all of which have corresponding concepts in Fluid's composition language. However, Fluid does not incorporate all features found in CoML. For example, CoML's somewhat imperative elements, such as property access and modification, as well as function calls and event triggering, were not included. Fluid's composition language instead focusses on the declarative description of components, their interconnections and configurations, and relies on component implementations to encapsulate runtime behaviour. Although a more mature version of the Fluid framework may incorporate imperative instructions, possibly via the use of inline scripting as supported by BML, the effect of OODDP manipulations on such elements has yet to be explored.

VISSION Despite its apparent support for component composition and configuration, VISSION's environment for component composition has a number of traits that clearly relate it to integration languages. For example, the methods used to expose DLL component properties are very similar to those used to expose C++ functions to higher level languages including Lua and Python[2]. Furthermore, VISSION's composition language is similarly used as a language environment for the integration of implementations: its metaclass concepts provide one-to-one

[2]For example, see LuaBind `http://www.rasterbar.com/products/luabind.html` and Boost's Python bindings `http://www.boost.org/libs/python/doc/index.html`.

mappings from component implementations to their interpreted counterparts. By contrast, Fluid's composition language supports higher level manipulations by hierarchically composing its small scale, highly configurable components into an object oriented type system. The type system supported by Fluid's composition language exists at a higher level of abstraction than the individual components, and the OODDP manipulations supported by the language are intended to make widespread changes to application composition and configuration.

Contigra Contigra's composition language has a very similar set of features to that supported by the Fluid framework. However, the scope of the latter project is ultimately limited by its underlying technologies. Much like the Fluid prototype, the current incarnation of Contigra is clearly targeting the visualisation of simple virtual environments. Meanwhile, the current evolution of the Fluid framework is intentionally more generalised, and is consequently able to apply its OODDP manipulations to a much broader context of software components and applications.

ChefMaster ChefMaster is just one example of a range of composition frameworks advocating the use of white or grey box components for the purpose of component adaptation. However, Fluid's emphasis on component customisation, along with the OODDP type system supported by its composition language, aim to increase component adaptability and to bypass the need to break the encapsulation of black box components. Furthermore, alternative methods for adapting components have been proposed, many of which rely on wrapping component interfaces in order to modify their external interfaces, provided functionality, and contextual requirements [116].

XCompose Although XCompose makes some use of XML for component specifications, much of its composition language appears to build imperative features upon the declarative XQuery language [138]. For example, XCompose's merge and inline composition pattern templates are written as sequences of XQuery statements that progressively form derived classes when executed. Although XCompose thus includes support for OODDP inheritance, such manipulations are exposed in the imperative language as functional operations, and require additional parameterisation to handle special cases. Composition pattern templates thus lack the expressiveness afforded by Fluid's composition language, where the inheritance relationship is part of a declarative syntax, and the semantics of OODDP manipulations are encapsulated by the Fluid framework. XCompose thus presents a lower level, more explicit, and less approachable language to its users.

PICCOLA The use of forms in PICCOLA has some similarity to Fluid's use of namespaces as a deployment concept, although the former are more widely used in PICCOLA, and are not only employed for deployment, but also take part in message passing, composition analysis, and other operations. Although certain concepts in Fluid were influenced by Lua's tables, PICCOLA's forms resemble them more closely.

There is a clear distinction between the Fluid framework, Fluid's composition language, and composition languages like PICCOLA. PICCOLA is an imperative, functional language, with support for multiple architectural styles and a focus on composition analysis. Despite its imperative syntax, PICCOLA's overall level of abstraction is much higher than that supported by Fluid, which focusses on the expressive manipulation of software structure and configuration. PICCOLA, and other languages with such a focus on higher level architectural issues, thus bridge the boundary between composition languages focussing on the wiring of software components, and ADLs intent on the description and analysis of software architecture.

VHD++ Despite its support for a wide range of high level concepts, many of which may be parameterised, the VHD++ framework does not make use of a data driven approach: there is no emphasis on data driving behaviour[3], and its composition lacks data manipulations such as inheritance relationships. Although VHD++ clearly demonstrates the use of its composition language to parameterise and compose both application software and content, the focus of such compositions is the wiring of these various elements, and not the influence of runtime behaviour.

The Fluid framework and composition language provide an alternative component technology to those discussed above. As is common among such technologies, the Fluid project bears a strong resemblance to its predecessors. In most cases, a component technology will extend another with a unique concept, relying on a proven approach to form the foundation for novel ideas. Meanwhile, the most effective or commonly supported ideas become increasingly prolific until they are accepted as cornerstones of the field. Consequently, almost all component models are based on the principles of components, script and glue, while a wide variety of compositional notations are supported by composition languages.

Fluid contributes to the evolution of CBSD by introducing a unique amalgamation of established approach component technologies and OODDP. Although a selection of the technologies discussed above include features and functionality overlapping those of the Fluid project, Fluid's framework and composition language maintain a unique approach to developing component based applications. Fluid makes use of small scale, data driven computational encapsulations, and provides a composition language supporting a range of OODDP definitions and manipulations.

[3]The references to data driven behaviour in [42] are in fact referring to event-based component connections and are not related to use of the term *data driven* as used throughout this work.

4.2.3 Architecture and methodology

ADLs support the description of component based solutions at higher level of abstraction than the composition languages discussed in Section 4.2.2, and often include support for more abstract concepts allowing the definition of compositions, their manipulations and their analysis. Consequently, there is little surprise to see OO inheritance relationship present in a number of ADLs, including C2 [139], Rapide [140], and Darwin [141]. Indeed, the following listing provides an example from ACME [142] demonstrating its use of OO inheritance to form a derived component type by inheriting, overriding, and extending the various elements of a parent type in a similar way to Fluid's OODDP manipulations.

```
Component Type FilterType =
{
        Ports { stdin; stdout; };
        Property throughput : int;
};
Component Type UnixFilterType extends FilterType with
{
        Port stderr;
        Property implementationFile : String;
};
```

There is a difference between the inheritance relationships expressed in ADLs such as ACME and those supported by the Fluid framework and its composition language. The first, and perhaps most prominent distinction is one of representation, particularly in the level of abstraction and granularity supported. ADLs concentrate on the architectural level, focussing on large scale components often representing concepts from the scale of objects to entire systems. Meanwhile, the type systems represented by ADLs are used to represent software architecture via component composition. Some of the relationships seen in ADLs, such as association, composition and generalisation may be found in high level notations such as UML 1.0, but ADLs also make widespread use of connectors including pipes, streams, and message passing.

By contrast, the Fluid project concentrates on highly configurable, small scale components forming *partial facets* of software objects. Fluid's facets are scaled up through incremental composition to the level of OO classes, where they become subject to OO relationships such as inheritance, and may be instantiated in order to form component applications. While ADLs are used to form software architectures, Fluid's composition language is used to form data driven software structures. Fluid's composition language and framework thus exhibit a lower level of abstraction than that supported by ADLs: its composition language is used to form low level component relationships, and disregards higher level architectural reasoning.

A further difference can be seen in the direction of information propagation throughout the application descriptions written using ADLs and Fluid's composition language. In ADLs, information is derived in a bottom up fashion: the atomic elements of ADLs are used to describe component specifications and compositions, which are in turn glued together to describe software architectures. Software architectures specified by ADLs are typically subjected to further analysis, which derives information pertaining to composition correctness, the flow of data through a software architecture during its execution, the quality of service that may be expected from its runtime behaviour, and so on. Such non-functional attributes of software architectures are often central to the use of ADLs, and their representation and analysis are commonly facilitated by ADL syntax and semantics.

On the other hand, the direction of information flow in Fluid compositions is intentionally top down: Fluid's composition language has been explicitly designed to provide an expressive environment for the composition and manipulation of small scale components and their parameterisations. Meanwhile, the Fluid framework facilitates the propagation of high level descriptions down to a lower level where data may drive the runtime behaviour of component implementations. From an alternative viewpoint, the lower tiers of the Fluid framework provide a deployment and intercommunication space for a large number of finely grained data driven components. Fluid's higher tiers, including its composition language, are responsible for proving an expressive notation for making widespread changes to application behaviour via data manipulations.

The Fluid framework makes use of a high level language and OODDP manipulations in order to drive the construction of a software structure consisting of computation encapsulations at a lower level of abstraction. In this way, Fluid bears some similarity to the MDSD approach, where high level notations describing software design are incrementally processed in order to ultimately derive corresponding implementations. Indeed, Fluid shares a focus on high level notations as seen in MDSD: in the Fluid framework, application description, along with the OODDP concepts it describes, forms part of the software structure and is used to determine overall application behaviour. Furthermore, the process of constructing software structures from higher level notations is similar to the process of code generation applied in model driven methodologies, particularly those constructing OO types from XML [97].

However, a distinction between Fluid and MDSD can be found in the overall direction of their constituent processes. Processes in MDSD are typically top down, with no loops or feedback, and no opportunity for lower level platform specific concepts to influence the higher level platform independent models. In contrast, Fluid component implementations are able to affect the overall software structure via naming system modifications. In addition, Fluid does not perform code generation, but instead relies on a fixed factory pattern that does not involve the use of a compilation step.

Despite the differences between the MDSD methodology and the Fluid approach to software development, there is clear potential for additional benefit to be introduced by using Fluid's higher level data descriptions to drive the *generation* of lower level implementations as well as their runtime behaviour. This is supported by existing work demonstrating the combination of model driven and component oriented technologies [55], and may be further facilitated by the use of metadata if a future migration to languages such as C♯ is undertaken [49]. The development of support for higher level concepts above those currently in use by the Fluid framework is an interesting avenue for further work.

While the high level behaviour of the Fluid framework forms links with MDSD, its use of small scale components defining *portions* of a given object's overall functionality also has some resemblance to the aspect oriented (AO) methodology. Fluid's small components could potentially be used to provide cross-cutting behaviours such as logging or security, and subsequently integrated in to a variety of OODDP types and instances. Due to their fine granularity, and their use to form more coarse and abstract encapsulations, Fluid components may seem attractive as aspects that may be woven by Fluid's composition model.

Although the Fluid framework may appear to support a number of AO technologies, its design and implementation have no native support for join point models. Fluid is therefore incapable of explicitly defining join points, pointcuts or the encapsulation of advice, although such concepts could be supported via the use of an appropriate component specification, application description, and component implementations, respectively. However, the resulting implementation would lose the semantics associated with AO, and would instead resemble an OODDP component deployment rather than its AO counterpart. Although the Fluid framework is not currently applicable to the AO methodology, recent research [35] highlights potential techniques and benefits for the combination of CBSD and AO technologies.

4.3 Computer Games Technology

Data driven programming, as seen in modern game technologies, was originally conceived as a means by which static OO hierarchies written in low level languages such as C++ could be replaced with a more dynamic type representation. The resulting data driven type systems could be easily modified in response to the ever-changing requirements of game designers, without the need to refactor crystallised class hierarchies. The increasing volumes and complexity of game content data has lead to the proliferation of data driven techniques, resulting in their widespread use today. Furthermore, the data driven methodology within the game development field continues to mature, as evidenced by a number of links with more mainstream CBSD concepts in more recent literature [95].

However, despite being widely used, and more recent associations with mainstream technologies, data driven methodologies remain limited to the definition of game content. With the notable exception of Microsoft's DirectX rendering API and its earlier input, sound and networking equivalents, which have always been deployed as COM components, game developers do not currently employ component based technologies in their game engines. The most likely reason for this is performance: game engines are often designed to be as efficient as possible, with certain routines being optimised in assembly code in order to take advantage of hardware-specific functionality. Furthermore, many of today's game engines are required to provide performance where it matters most, while application flexibility is delivered by data driven techniques.

The Fluid framework's incorporated OODDP technologies are similar to those used in the games industry, as introduced in Section 2.3.1. However, their combination with CBSD from mainstream computer science introduces a number of important distinctions, as discussed below.

Alex Duran allows the Fluid framework's use of OODDP to be placed in the context of other OODDP implementations. For example, it is clear that Fluid makes use of an object based representation for its various OODDP abstractions, by supporting an explicit hierarchy between OODDP objects and their constituent parts. In contrast, a component based representation[4] reverses the ownership relationship present in Fluid by requiring leaf elements of the object hierarchy to maintain a reference to their parent objects.

Fluid also supports abstractions for OODDP types, templates and instances, as well as OO inheritance relationships between them. The Fluid framework's OODDP representations form part of its component model, and play a key role in its composition language.

Furthermore, Fluid components are able to form dependencies on other components by connecting the exported functionality of one component to the contextual requirements of another. Fluid's inter-component connections thus resemble the component requirements discussed by Duran, albeit at a finer level of granularity. However, Fluid does not currently enforce component exclusion, which requires that a component of a given type is *not* present as part of a component topology.

The Fluid framework does not make a distinction between simple and complex components, which encapsulate concepts at the scales of program state to behavioural abstractions respectively. Indeed, the Fluid framework is capable of incorporating components at a wide range of scales, from single values supported by Value objects, through to more conventional components corresponding to OO libraries.

[4]The term *component based* here refers to its use in OODDP rather than its use in computer science.

Duran also presents a number of concepts that are not supported by the Fluid framework, such as the various optimisations available to OODDP implementations. However, such optimisations may be introduced as the Fluid framework matures.

Scott Bilas Bilas' system motivated the use of OODDP concepts as part of the Fluid prototype, and greatly influenced its design and implementation. However, the Fluid framework makes a number of improvements in order to increase the flexibility of its OODDP abstractions. For example, the OODDP types in Bilas' system and the Fluid prototype are built upon a number of low level component types that form part of the underlying framework. In the Fluid framework, all components are developed independently, and are not restricted by the interfaces or semantics introduced by lower level functionality. Furthermore, Fluid's component model and composition language form inter-component connections at a higher level of abstraction than Bilas' automated library linker. Whereas Bilas' function binding tool relies upon a range of low level and both platform and compiler dependent properties, the concepts supported by Fluid's component model and composition language are more scalable and flexible to future modification.

Chris Stoy Although Stoy's system bears some resemblance to the OODDP concepts incorporated by the Fluid framework, the introduction of a component based approach makes the Fluid framework much more flexible: Stoy's `GameObjects` communicate with one another by accessing `GameObjectComponent` instances belonging to other `GameObjects` via a common base type, and then downcasting the `GameObjectComponent` instance to its concrete type for manipulation. This type of interaction between components requires that component implementations are aware of the class types representing other components at compile time. Consequently, component implementations remain tightly coupled. In contrast, Fluid's components are connected via the framework's component model and unified type system. Fluid's components may be independently developed and are not affected by modifications to other components' implementations.

Bjarne Rene The higher level flexibility of Rene's OODDP implementation remains unclear. Component message types are enumerated at design time and represented using constant values, which also appear in component implementations. These fixed values could form a tight coupling between communicating component implementations. In the Fluid framework, messages consist of direct connections between publishing and subscribing instances, with the framework's component model and low level providing the necessary connections. The Fluid framework thus provides a more flexible component communication system.

Fluid successfully applies the data driven methodology in its OODDP framework and composition language, forming explicit links between data driven programming and mainstream CBSD. In contrast to its use in modern game engines, the Fluid framework incorporates an OODDP composition language in order to support the composition of small scale computational encapsulations, delivering both scene content *and* overall application behaviour. Although this work is unlikely to impact the game development community as a whole, the Fluid framework demonstrates the potential for broader applications of game technologies such as OODDP.

4.4 Summary

Figure 4.3 provides an illustrative summary of the contextualisation of the Fluid project.

Figure 4.3a presents the limitations of a number of related technologies *Data centric* applications are those where data plays a central role in the application's functionality. For example, the functionality of GIS is based on the selection of an appropriate data model, which determines the meaningful representation, storage, communication and manipulation of geographically referenced data. However, the runtime behaviour of data centric applications is typically static, and cannot be changed without modifying and recompiling the software implementation.

In contrast, the behaviour of *data driven* applications may be influenced by their data inputs. DDDAS incorporate a synergistic feedback loop between simulation and measurement, where measurement data informs the simulation, which in turn drives further measurements. However, the extent of data driven behaviour in DDDAS is typically limited to the modification of control flow and simulation parameterisation. DDDAS are not currently able to dynamically alter the *capabilities* of their software implementations, and the current level of adaptation exhibited by DDDAS is accommodated by their otherwise static software implementations.

Component based approaches provide a range of solutions for manipulating software capabilities using high level composition languages. However, the CBSD field is currently immature, with a wide range of independent projects failing to converge in order to form supportive standards. Furthermore, the flexibility of component applications is typically provided by component manipulations at a high level of abstraction, often making use of component wrappers, and there is little data driven behaviour or configurable flexibility provided by components themselves.

OODDP technologies are becoming increasingly popular in contemporary game titles, and support the manipulation of application functionality and behaviour using high level notations embedded in game content data. OODDP focuses on creating a wide variety of immersive experiences through the use of highly configurable small scale components. However, the application of OODDP is currently limited to defining game content, and its use in the more general context of software development has yet to be explored.

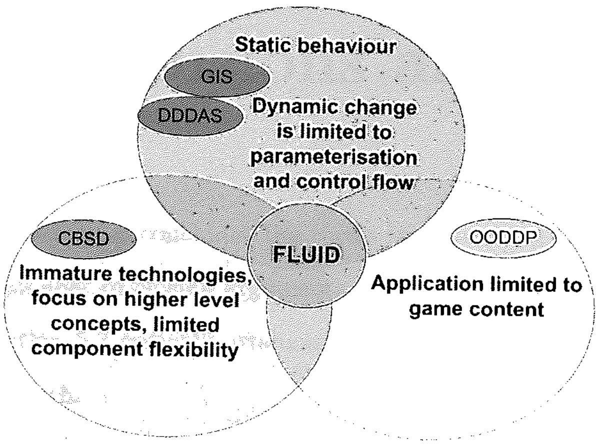
Figure 4.3b shows how these technologies have formed the basis of the work described in this thesis The Fluid project combines CBSD with OODDP in order to define a novel component framework and composition language supporting a range of OODDP concepts and manipulations, and focussing on the use of small scale data driven black box components. Consequently, the Fluid project builds upon a wide range of research from the CBSD field in order to define the design and implementation of the Fluid framework's various component technologies. A number of articles and presentations related to game developers' use of data driven approaches have also influenced the work discussed in this thesis.

While the aim of the Fluid project is to successfully and effectively combine technologies from the CBSD and game development fields, GIS and DDDAS provide the motivation for much of the work presented in Chapter 3, and also supply the context of any example applications.

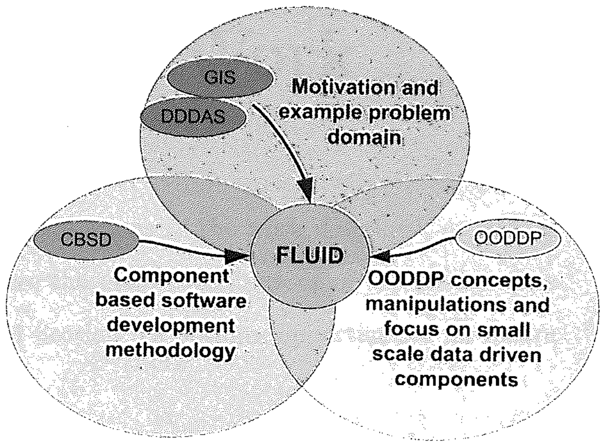
Figure 4.3c gives the contributions made by the Fluid project The Fluid project contributes to CBSD by defining novel component based technologies emphasising the use of fine grained data driven components. While a small number of studies show potential benefits of small components [120, 119, 118], their use has yet to be fully explored by the CBSD research community.

The Fluid framework introduces a number of potential implications for game developers. This thesis demonstrates that a hybrid approach making use of CBSD and OODDP has broader applications beyond the definition of game content. Indeed, OODDP has a number of potential uses in introducing higher level abstractions to describing the selection, interconnection and configuration of component based applications. There may also be benefits to incorporating component based approaches as part of more flexible and extensible game engines.

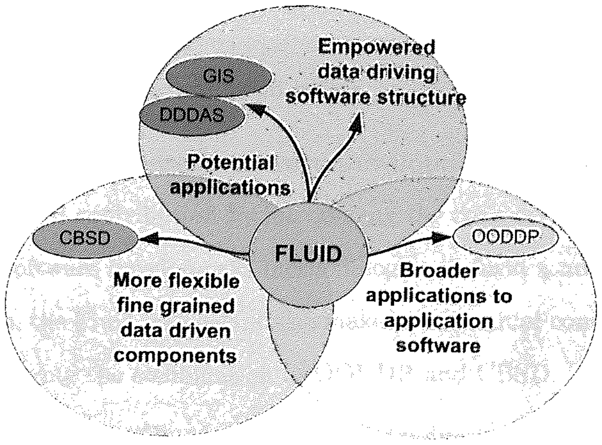
Finally, Section 4.1 has discussed potential applications of the Fluid approach to DDDAS, where an OODDP based component model and composition language could form part of a more flexible modelling and measurement process. A speculative Fluid based DDDAS would be more flexible to change by further empowering data in order to determine application capabilities as well as its runtime behaviour.


Static behaviour
GIS
DDDAS
Dynamic change is limited to parameterisation and control flow
CBSD
FLUID
OODDP
Immature technologies, focus on higher level concepts, limited component flexibility
Application limited to game content

(a) Current limitations of related technologies


GIS
DDDAS
Motivation and example problem domain
CBSD
FLUID
OODDP
Component based software development methodology
OODDP concepts, manipulations and focus on small scale data driven components

(b) Technologies forming the basis of the Fluid project


GIS
DDDAS
Empowered data driving software structure
Potential applications
CBSD
FLUID
OODDP
More flexible fine grained data driven components
Broader applications to application software

(c) The contributions of the Fluid project

Figure 4.3: A summary of the limitations and influential aspects of the technologies related to the Fluid project, and an illustration of its contributions.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis by presenting the contributions of the work described, as well as a number of closing arguments. Section 5.1 summarises the main contributions of the thesis, while Section 5.2 critically evaluates the work described and Section 5.3 outlines opportunities for future research.

5.1 Contributions

The principal contributions of this thesis are listed below.

A Fluid approach to component based software development The work described in Chapter 3 makes a **conceptual** contribution by introducing the data driven programming paradigm to component based software development methodologies to form a novel combination of technologies. Furthermore, the Fluid framework also makes a **technical** contribution by successfully implementing and applying the combination of OODDP and CBSD.

The Fluid framework is the result of successfully introducing the data driven programming paradigm to component based software development methodologies. The Fluid approach to CBSD emphasises flexibility and extensibility through the use of small scale, highly configurable components exhibiting a variety of data driven behaviour. Furthermore, the definition of Fluid applications is facilitated by a declarative composition language supporting an OO programming style via its incorporation of OODDP concepts and manipulations. The Fluid framework has been successfully applied to model a simple but representative example, as discussed in Section 3.3.

A Fluid approach to DDDAS The thesis makes a **conceptual** contribution to the field of DDDAS by proposing a novel combination of OODDP, CBSD and DDDAS technologies with the potential to overcome certain limitations of current DDDAS. However, the technical application of the proposed solution lies outside the scope of this work.

DDDAS currently incorporate measurement data into an evolving model, which in turn drives further measurement. The synergistic feedback loop between measurement and model allow DDDAS to focus resources on measuring more effectively. However, this thesis identified a limitation of current approaches; while the runtime behaviour of DDDAS may be driven by data, the extent of influence available to data is in fact limited to modifying application control flow and adjusting the parameterisation of methods.

This thesis has speculated a number of potential applications of the Fluid framework within the DDDAS field of research. Section 4.1 has suggested how a combination of Fluid and DDDAS technologies would allow the functionality of DDDAS to change according to measurement data. The proposed combination of Fluid and DDDAS technologies would be particularly suitable for simulations concerning discrete phenomena, where Fluid's OODDP concepts are most effective.

A Fluid approach to data driven rendering The work described in Section 3.1 builds upon a number of concepts that have been recently introduced by Microsoft. However, the rendering system described in a related publication [3] makes a **technical** contribution by implementing a platform independent alternative to an upcoming version of the Direct3D library, making use of open source and third party technologies.

As described in Section 3.1, the Fluid prototype's render system provides a data driven rendering pipeline, which encompasses functionality ranging from high-level SAS manipulations to low level OpenGL API calls. Fluid's render system implementation makes use of a range of third party solutions that collectively deliver a platform independent alternative to an anticipated Direct3D release. The PirateHat rendering library encapsulates an implementation of the Fluid prototype's rendering system that may be used independently of the Fluid project [3]; it is the first rendering library to offer a complete data driven rendering pipeline supporting SAS instructions to the OpenGL platform.

5.2 Critical Evaluation

Summarising Chapter 4, the key strengths and weaknesses of the Fluid project are:

Successful combination of technologies CBSD and OODDP have been successfully combined in the Fluid framework, a C++ component framework supporting an object oriented data driven component model and composition language. Fluid's composition language provides a number of

OO abstractions and manipulations to define Fluid applications using fine grained components with data driven runtime behaviour.

Successful application of the Fluid framework The Fluid framework has been successfully applied to model a simple traffic simulation. Although the example described in Section 3.3 is very basic, the application demonstrates the use of the Fluid framework, including its low level functionality, component model and composition language, to define a complete component based application through the combination of CBSD and OODDP technologies.

Enhanced parameterisation The Fluid framework supports the definition of component applications using fine grained, highly configurable black box components. Through the combination of its component model and OODDP composition language, the Fluid framework allows fine grained control and configuration of computational abstractions. Fluid applications are thus able to incorporate a greater degree of parameterisation and data driven behaviour than typically seen in composition languages, without the need to break component encapsulation.

Data driven rendering The PirateHat rendering library [3] encapsulates a data driven rendering pipeline that can be manipulated using a high level scripting language. The PirateHat library delivers similar functionality to that anticipated from an upcoming version of the Direct3D API, but uses open third party solutions in order to remain platform independent.

Low level implementation Although the Fluid framework augments the low level functionality of the C++ implementation language, the concepts provided by the Fluid framework exhibit a low level of abstraction, which may limit their use in the development of Fluid components. While the Fluid framework could be improved to support higher level abstractions, the desired functionality is already provided by a number of software development platforms more suited to CBSD, most notably JavaBeans and the .NET framework. It may be beneficial to migrate the Fluid framework to a more appropriate development platform.

Overly verbose While the successful combination of CBSD and OODDP allows Fluid's application configurations to make use of a range of OO concepts and manipulations, the definition and modification of Fluid's XML based component specifications and application configurations may be difficult without the aid of visual tools. The difficulty of manipulating Fluid's XML documents is partly due to the XML schema in use. An improved XML schema, the provision of appropriate visual tools, and a composition supporting higher level functionality such as the introduction of new type descriptions, are likely to improve the expressiveness of Fluid's XML documents.

Unknown scalability Section 3.3 demonstrates the application of the Fluid framework to model a simple traffic simulation. However, the effectiveness of Fluid's CBSD approach when modelling larger, more complex systems remains untested.

5.3 Future work

The Fluid project has shown that the combination of CBSD and OODDP offer an effective platform for the definition of flexible component based applications. However, the current design and implementation of the Fluid framework are immature, and further iterations would be required before the research community would be able to make practical use of Fluid technologies. The following paragraphs provide a brief overview of interesting opportunities for further research within the context of the Fluid framework.

Applications in DDDAS A number of particularly interesting opportunities for further research are provided by the continued integration of DDDAS and Fluid technologies. Chapter 4 describes a range of potential applications. However, the Fluid framework is currently unable to fully support such use, due to the additional requirements of DDDAS and the complex nature of their implementations. A speculative Fluid based DDDAS would be most effective if its simulation and modelling software could evolve over time in response to changing requirements, thus focussing computational resources in the same way as the measurement process is currently optimised. In order to facilitate this, the Fluid framework would have to incorporate a range of additional functionality.

Although Fluid applications are presently open to the modification of OODDP instances after their initial deployment, the current framework implementation has limited support for maintaining the correctness of such changes. While a number of research projects explore the runtime adaptation of component based applications, the granularity and data driven nature of Fluid's components may limit the applicability of their solutions. For example, while many component frameworks maintain a one to one correlation between their components and those concepts present in their composition languages, Fluid's OODDP abstractions may be used to collect many small components into a single unit of deployment. Furthermore, while the use of OODDP concepts has proven to be effective for the initial definition of Fluid applications, appropriate notations for the adaptation of OODDP types, instances, deployments and parameterisations have yet to be realised.

The need to modify Fluid applications presents a number of interesting questions and opportunities for its composition language. Should application change be driven by Fluid's composition language, or should such changes be made directly via Fluid's component model or its lower tiers? If Fluid's composition language is to include the specification of application change, how should such changes be represented? Early designs for the Fluid framework provided support for the Lua scripting language throughout its fundamental systems, component model and composition language. The intention of these designs was to allow application developers to describe a range of application behaviour using a high level scripting language. However, Lua integration was not realised due to time limitations. Scripting languages have been made available in a number of composition languages including BML [37]. An alternative approach is provided by CoML, which supports a range of platform independent behavioural operations as part of its XML syntax [33].

Higher level abstractions Further development of the Fluid framework would also provide an opportunity for building higher level abstractions upon those already supported. Incrementally abstract concepts and manipulations could potentially allow the specification, selection, interconnection and parameterisation of Fluid components to be defined with increasing flexibility and expressiveness.

Of particular interest are those abstractions facilitating visual programming. The initial development of OODDP methodologies in games was highly motivated by the need to expose software structure to game designers via tools. Consequently, a vast array of visual tools have been developed by game developers and consumer communities for many data driven game titles. Furthermore, many game development studios are releasing tools alongside their games, so that lay consumers are able to design their own content, stories and player experiences. Meanwhile, many component based research projects are also producing visual tools for the development of component applications. For example, SuperGlue [34], Contigra [39] and VISSION [38] support the interactive assembly of component implementations using visual interfaces. As the Fluid project combines OODDP technologies with those from CBSD, development of appropriate visual tools should be straightforward.

Higher level abstractions may also allow the Fluid project to participate in model driven approaches to software development; Fluid's OODDP concepts and manipulations already bear some resemblance to some of the relationships used in UML static class diagrams. Further research in this area may lead to a model driven approach making use of Fluid's higher level abstractions to express its platform independent representations, while a range of Fluid components provide platform specific implementations. An interesting question in this area of research is whether a small selection of finely grained, highly configurable components can provide a palette of computation encapsulations that may be used in place of automatically generated concrete code. If so, then Fluid's component model and composition language could provide appropriate mappings for a range of OO descriptions.

A potentially exciting outcome of developing increasing layers of abstraction upon those concepts already available to Fluid may be a further generation of empowered data formats that are completely self describing. Data written using such formats may be able to describe appropriate reading, processing, analysis and visualisation operations, using platform independent notations that drive the generation of corresponding software applications.

References

[1] S. H. Muggleton, "Exceeding human limits," *Nature*, vol. 440, pp. 409–410, 2006.

[2] A. Szalay and J. Gray, "Science in an exponential world," *Nature*, vol. 440, pp. 413–414, 2006.

[3] A. Jones, C. Mantle, and D. Cornford, "Data driven graphical applications: A fluid approach," in *Theory and Practice of Computer Graphics Eurographics UK Chapter Proceedings* (I. S. Lim and D. Duce, eds.), pp. 187–194, Eurographics Association, 2007.

[4] D. Treffry and S. Ferguson, eds., *Collins English Dictionary.* HarperCollins Publishers, 2006.

[5] B. McLaughlin, *Java and XML data binding.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[6] A. Brookes, "XML data binding," *Dr. Dobb's Journal*, vol. 28, pp. 26, 28, 30, 32, 35–36, March 2003.

[7] F. Atanassow, D. Clarke, and J. Jeuring, "UUXML: A type-preserving XML schema-Haskell data binding," in *Practical Aspects of Declarative Languages*, vol. 3057/2004 of *Lecture Notes in Computer Science*, pp. 71–85, Springer Berlin / Heidelberg, May 2004.

[8] M. Worboys and M. Duckham, *GIS: a computing perspective*, ch. 1, pp. 1 – 33. CRC Press, 2 ed., 2004.

[9] M. Worboys and M. Duckham, *GIS: a computing perspective*, ch. 8.3, pp. 305 – 316. CRC Press, 2 ed., 2004.

[10] S. Su, W. Sherman, F. Harris, and M. Dye, "TAVERNS: Visualization and manipulation of GIS data in 3D large screen immersive environments," in *International Conference on Artificial Reality and Telexistence Workshops*, pp. 656–661, IEEE Computer Society, 2006.

[11] J. Döllner and K. Hinrichs, "An object-oriented approach for integrating 3D visualization systems and GIS," *Computers and Geosciences*, vol. 26, no. 1, pp. 67–76, 2000.

[12] J. Metze, B. Neidhold, and M. Wacker, "Towards a general concept for distributed visualisation of simulations in virtual reality environments," in *Proceedings of the 9th International Workshop on Immersive Projection Technology* (E. Kjems and R. Blach, eds.), (Aalborg, Denmark), pp. 79–90, Eurographics Association, 2005.

[13] S. D. Bergen, R. J. McGaughey, and J. L. Fridley, "Data-driven simulation, dimensional accuracy and realism in a landscape visualization tool," *Landscape and Urban Planning*, vol. 40, pp. 283–293, May 1998.

[14] D. V. Pullar and M. E. Tidey, "Coupling 3Ds visualisation to qualitative assessment of built environment designs," in *Landscape and Urban Planning*, vol. 55, pp. 29–40, June 2001.

[15] K. Appleton and A. Lovett, "GIS-based visualisation of development proposals: reactions from planning and related professionals," *Computers, Environment and Urban Systems*, vol. 29, pp. 321–339, May 2005.

[16] K. Appleton and A. Lovett, "GIS-based visualisation of rural landscapes: defining 'sufficient' realism for environmental decision-making, landscape and urban planning," *Landscape and Urban Planning*, vol. 65, pp. 117–131, October 2003.

[17] E. S. Raymond, *The Art of Unix Programming*, ch. 9, p. 250. Addison-Wesley, September 2003.

[18] F. Darema, "Grid computing and beyond: the context of dynamic data driven applications systems," in *Proceedings of the IEEE*, vol. 93, pp. 692 – 697, March 2005.

[19] A. Chaturvedi, A. Mellema, S. Filatyev, and J. Gore, "DDDAS for fire and agent evacuation modeling of the Rhode Island nightclub fire," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 433–439, Springer, 2006.

[20] C. C. Douglas, J. D. Beezley, J. Coen, D. Li, W. Li, A. K. Mandel, J. Mandel, G. Qin, and A. Vodacek, "Demonstrating the validity of a wildfire DDDAS," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 522–529, Springer, 2006.

[21] R. Fujimoto, R. Guensler, M. Hunter, H. K. Kim, J. Lee, J. Leonard II, M. Palekar, K. Schwan, and B. Seshasayee, "Dynamic data driven application simulation of surface transportation systems," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 425–432, Springer, 2006.

[22] C. C. Douglas, J. C. Harris, M. Iskandarani, C. R. Johnson, R. J. Lodder, S. G. Parker, M. J. Cole, R. Ewing, Y. Efendiev, R. Lazarov, and G. Qin, "Dynamic contaminant identification in water," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 393–400, Springer, 2006.

[23] P. Kruchten, *The Rational Unified Process: An Introduction.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] R. Wuyts and S. Ducasse, "Composition languages for black-box components," in *Object Oriented Programming, Systems, Languages, and Applications* (D. H. Lorenz and V. C. Sreedhar, eds.), pp. 33–36, Technical Report NU-CSS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115, 2001.

[25] J.-G. Schneider and O. Nierstrasz, "Components, scripts and glue," in *Software Architectures — Advances and Applications* (L. Barroca, J. Hall, and P. Hall, eds.), pp. 13–25, Springer-Verlag, 1999.

[26] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz, "Piccola — a small composition language," in *Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches* (H. Bowman and J. Derrick, eds.), pp. 403–426, Cambridge University Press, 2001.

[27] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ch. 1, pp. 3 - 15. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ch. 8, p. 41. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[29] M. Büchi and W. Weck, "A plea for grey-box components," in *Proceedings of the 1st Workshop on the Foundations of Component-Based Systems* (G. T. Leavens and M. Sitaraman, eds.), pp. 39–49, September 1997.

[30] O. Nierstrasz and T. D. Meijler, "Requirements for a composition language," in *Object-Based Models and Languages for Concurrent Systems* (P. Ciancarini, O. Nierstrasz, and A. Yonezawa, eds.), vol. 924, pp. 147–161, Springer-Verlag, 1995.

[31] N. Jefferson and S. Riddle, "Towards a formal semantics of a composition language," in *Proceedings of the 3rd International Workshop on Composition Languages*, (Darmstadt,Germany.), July 2003.

[32] Y. D. Liu and S. F. Smith, "A formal framework for component deployment," in *Object Oriented Programming, Systems, Languages, and Applications*, (New York, NY, USA), pp. 325–344, ACM Press, 2006.

[33] D. Birngruber, "CoML: Yet another, but simple component composition language," in *Proceedings of the 1st Workshop on Composition Languages*, (Vienna, Austria), September 2001.

[34] S. McDirmid, "Interactive component assembly with SuperGlue," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, 2006.

[35] M. Bynens and W. Joosen, "On the benefits of using aspect technology in component-oriented architectures," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, 2006.

[36] D. Birngruber and M. Hof, "Using plans for specifying preconfigured bean sets," in *Technology of Object Oriented Languages and Systems* (Q. Li, D. Firesmith, R. Riehle, and B. Meyer, eds.), pp. 217–226, IEEE Computer Society, 2000.

[37] S. Weerawarana, F. Curbera, M. J. Duftler, D. A. Epstein, and J. Kesselman, "Bean Markup Language: A composition language for JavaBeans components," in *Conference on Object Oriented Technologies and Systems*, pp. 173–188, USENIX, 2001.

[38] A. Telea, "A component-based dataflow framework for simulation and visualization," in *European Conference on Object Oriented Programming Workshops* (A. M. D. Moreira and S. Demeyer, eds.), vol. 1743 of *Lecture Notes in Computer Science*, p. 187, Springer, 1999.

[39] R. Dachselt, M. Hinz, and K. Meißner, "Contigra: an XML-based architecture for component-oriented 3D applications," in *Proceeding of the 7th international conference on 3D Web technology*, (New York, NY, USA), pp. 155–163, ACM Press, 2002.

[40] D. J. Ram and C. Babu, "Chefmaster: An augmented glue framework for dynamic customization of interacting components," in *Proceedings of the 3rd International Workshop on Composition Languages*, (Darmstadt,Germany.), July 2003.

[41] N. Tansalarak and K. T. Claypool, "XCompose: An XML-based component composition framework," in *Proceedings of the 3rd International Workshop on Composition Languages*, 2003.

[42] M. Ponder, *Component-based methodology and development framework for virtual and augmented reality systems.* PhD thesis, IC Facult informatique et communications, 2004.

[43] J.-G. Schneider and J. Han, "Components – the past, the present, and the future," in *Proceedings of the 9th International Workshop on Component Oriented Programming* (W. W. Clemens Szyperski and J. Bosch, eds.), (Oslo, Norway), June 2004.

[44] S. D. Panfilis and A. J. Berre, "Open issues and concerns on component-based software engineering," in *Proceedings of the 9th International Workshop on Component Oriented Programming*, (Oslo, Norway), June 2004.

[45] H. van Vliet, *Software Engineering: Principles and Practice*, ch. 3.1, pp. 49–51. John Wiley and Sons, Ltd., 2 ed., 2000.

[46] H. van Vliet, *Software Engineering: Principles and Practice*, ch. 3.6, pp. 62–63. John Wiley and Sons, Ltd., 2 ed., 2000.

[47] E. Renaux and E. Lefebvre, "Component based method for enterprise application design," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, (Nantes, France), July 2006.

[48] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien, "A safe aspect-oriented programming support for component-oriented programming," in *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP06)*, vol. 200611 of *Technical Report*, (Nantes, France), Karlsruhe University, jul 2006.

[49] R. Rouvoy and P. Merle, "Leveraging component-oriented programming with attribute-oriented programming," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, vol. 200611 of *Technical Report*, (Nantes, France), Karlsruhe University, July 2006.

[50] S. Lopes, A. Tavares, J. L. Monteiro, and C. A. Silva, "Describing framework static structure : promoting interfaces with UML annotations," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, 2006.

[51] A. Scherp and S. Boll, "A lightweight process model and development methodology for component frameworks," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, (Glasgow, Scotland), 2005.

[52] S. Overhage, "Towards a standardized specification framework for component development, discovery, and configuration," in *Proceedings of the 8th International Workshop on Component Oriented Programming* (J. Bosch, C. Szyperski, and W. Weck, eds.), 2003.

[53] F. Dominguez-Mateos and R. Hijon-Neira, "An architectural component-based model to solve the heterogeneous interoperability of component-oriented middleware platforms," in *Proceedings*

of the 11th International Workshop on Component Oriented Programming, (Nantes, France), July 2006.

[54] L. A. Gayard, P. A. de Castro Guerra, A. E. de Campos Lobo, and C. M. F. Rubira, "Automated deployment of component architectures with versioned components," in *Proceedings of the 11th International Workshop on Component Oriented Programming*, (Nantes, France), July 2006.

[55] S.-Y. Lee, O.-C. Kwon, M.-J. Kim, and G.-S. Shin, "Research on an MDA based COP approach," in *Proceedings of the 8th International Workshop on Component Oriented Programming*, (Darmstadt, Germany), July 2003.

[56] O. Nano and M. Blay-Fornarino, "Using MDA to integrate services in component platforms," in *Proceedings of the 8th International Workshop on Component Oriented Programming*, (Darmstat, Germany), July 2003.

[57] H. Koziolek and S. Becker, "Transforming operational profiles of software components for quality of service predictions," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, 2005.

[58] J. Happe and V. Firus, "Using stochastic petri nets to predict quality of service attributes of component-based software architectures," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, (Glasgow, Scotland), July 2005.

[59] M. Meyerhöfer and F. Lauterwald, "Towards platform-independent component measurement," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, (Glasgow, Scotland), July 2005.

[60] R. Reussner, I. Poernomo, and H. Schmidt, "Contracts and quality attributes for software components," in *Proceedings of the 8th International Workshop on Component Oriented Programming* (W. Weck, J. Bosch, and C. Szyperski, eds.), 2003.

[61] A. Alvaro, E. S. Almeida, and S. L. Meira, "Quality attributes for a component quality model," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, (Glasgow, Scotland), 2005.

[62] F. Puntigam, "In components we trust - programming language support for weak protection," in *Proceedings of the 10th International Workshop on Component Oriented Programming*, (Glasgow, UK), July 2005.

[63] A. Zeid, M. Messiha, and S. Youssef, "Applicability of component-based development in high-performance systems," in *Proceedings of the 9th International Workshop on Component Oriented Programming* (J. Bosch, C. Szyperski, and W. Wolfgang, eds.), June 2004.

[64] I. Crnkovic, "Component-based approach for embedded systems," in *Proceedings of the 9th International Workshop on Component Oriented Programming*, 2004.

[65] T. Parsons and J. Murphy, "A framework for detecting, assessing and visualizing performance antipatterns in component based systems," in *Object Oriented Programming, Systems, Languages, and Applications*, pp. 316 - 317, 2004.

[66] I. Sommerville, *Software Engineering*, ch. 11, pp. 241–265. International Computer Science, Pearson Education Limited, 8 ed., 2007.

[67] P. Kogut and P. Clements, "Features of architecture description languages." Draft of a Carnegie-Mellon University/Software Engineering Institute Technical Report, 1994.

[68] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," in *Proceedings of the 6th European Software Engineering Conference*, (Zürich, Switzerland), pp. 60–76, September 1997.

[69] A. D. Fuxman, "A survey of architecture description languages." Technical report CSRG-407, Department of Computer Science, University of Toronto, Canada, 2000.

[70] K. Wallnau, J. Stafford, S. Hissam, and M. Klein, "On the relationship of software architecture to software component technology," in *Proceedings of the 6th International Workshop on Component Oriented Programming*, (Budapest, Hungary), June 2001.

[71] J. Oberleitner and T. Gschwind, "Requirements for an architectural composition language," in *Proceedings of the 7th International Workshop on Component Oriented Programming*, 2002.

[72] J. Matevska-Meyer, W. Hasselbring, and R. Reussner, "Software architecture description supporting component deployment and system runtime reconfiguration," in *Proceedings of the 9th International Workshop on Component Oriented Programming* (J. Bosch, C. Szyperski, and W. Wolfgang, eds.), June 2004.

[73] S. Zhang and S. Goddard, "xSADL: An architecture description language to specify component-based systems," in *Proceedings of the International Conference on Information Technology: Coding and Computing*, vol. 2, pp. 443–448, 2005.

[74] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[75] Y. Ledru, R. Sanlaville, and J. Estublier, "Defining an architecture description language for dassault systémes," in *Proceedings of the 4th International Software Architecture Workshop* (B. Balzer and H. Obbink, eds.), pp. 115 – 120, June 2000.

[76] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, pp. 70–93, January 2000.

[77] B. Selic, "UML 2: A model-driven development tool," *IBM Systems Journal*, vol. 45, no. 3, pp. 607–620, 2006.

[78] M. McGuire, "Efficient shadow volume rendering," in *GPU Gems* (R. Fernando, ed.), ch. 9, pp. 137–166, Boston, MA: Addison Wesley, 1 ed., 2004.

[79] S. Kozlov, "Perspective shadow maps: Care and feeding," in *GPU Gems* (R. Fernando, ed.), ch. 14, pp. 217–244, Boston, MA: Addison Wesley, 1 ed., 2004.

[80] U. Assarsson, M. Dougherty, M. Mounier, and T. Akenine-Moller, "An optimized soft shadow volume algorithm with real-time performance," in *Proceedings of the Annual ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (W. Mark and A. Schilling, eds.), (Aire-la-ville, Switzerland), pp. 33–40, Eurographics Association, July 2003.

[81] S. Brabec and H.-P. Seidel, "Shadow volumes on programmable graphics hardware," *Computer Graphics Forum*, vol. 22, no. 3, pp. 433–440, 2003.

[82] L. Williams, "Casting curved shadows on curved surfaces," in *Computer Graphics*, vol. 12, pp. 270–274, SIGGRAPH, August 1978.

[83] M. Stamminger and G. Drettakis, "Perspective shadow maps," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 557–562, SIGGRAPH, ACM Press, 2002.

[84] L. Cenydd and W. Teahan, "The dynamic animation of ambulatory arthropods," in *Theory and Practice of Computer Graphics* (I. S. Lim and D. Duce, eds.), pp. 21 – 28, Eurographics Association, June 2007.

[85] O. E. Gundersen and L. Tangvald, "Level of detail for physically based fire," in *Theory and Practice of Computer Graphics* (I. S. Lim and D. Duce, eds.), pp. 21 – 28, Eurographics Association, June 2007.

[86] A. Herwig and P. Paar, *Trends in GIS and Virtualization in Environmental Planning and Design*, ch. Game Engines: Tools for Landscape Visualization and Planning?, pp. 162–171. Wichmann, 2002.

[87] D. Fritsch and M. Kada, "Visualisation using game engines," in *Geo-Informations-Systeme*, vol. 2004, pp. 32–36, June 2004.

[88] B. Kot, B. Wuensche, J. Grundy, and J. Hosking, "Information visualisation utilising 3D computer game engines case study: a source code comprehension tool," in *Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction*, (New York, NY, USA), pp. 53–60, ACM Press, 2005.

[89] D. Rohrl, "Two dozen ways to screw up a perfectly good project," in *Game Developers Conference Proceedings*, 2002.

[90] N. Llopis, "By the books: Solid software engineering for games," in *Game Developers Conference Proceedings*, 2003.

[91] G. Booch, "Best practices in game development," in *Game Developers Conference Proceedings*, 2006.

[92] S. Bilas, "A data-driven game object system," in *Game Developers Conference Proceedings*, 2002.

[93] A. Duran, "Building object systems - features, tradeoffs, and pitfalls," in *Game Developers Conference Proceedings*, 2003.

[94] S. Bilas, "Fubi: Automatic function exporting for networking and scripting," in *Game Developers Conference Proceedings*, 2001.

[95] C. Stoy, "Game object component system," in *Game Programming Gems 6* (M. Dickheiser, ed.), ch. 4.6, pp. 393–403, Rockland, MA, USA: Charles River Media, Inc., February 2006.

[96] B. Rene, "Component based object management," in *Game Programming Gems 5* (K. Pallister, ed.), ch. 1.3, pp. 25–37, Charles River Media, February 2005.

[97] E. van der Vlist, "XML driven classes in Python," in *Proceedings of the O'Reilly Open source Convention*, O'Reilly, July 2004.

[98] M. D. McCool, Z. Qin, and T. S. Popa, "Shader metaprogramming," in *Proceedings of the 17th Eurographics/SIGGRAPH workshop on graphics hardware* (S. N. Spencer, ed.), (New York), pp. 57–68, ACM Press, Sept. 1–2 2002.

[99] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, 2003.

[100] T. Reeves, D. Cornford, M. Konecny, and J. Ellis, "Modelling geometric rules in object based models: An XML/GML approach," in *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling* (A. Riedl, W. Kainz, and G. Elmes, eds.), pp. 133–148, Springer-Verlag, 2006.

[101] V. Mencl, "Autonomous points in component composition," in *Object Oriented Programming, Systems, Languages, and Applications*, (Tampa, FL, USA), pp. 83–84, October 2001.

[102] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: safe and predictable dynamic software updating," in *Principles of Programming Languages*, (New York, NY, USA), pp. 183–194, ACM Press, 2005.

[103] S. Zachariadis and C. Mascolo, "The satin component system-a metamodel for engineering adaptable mobile systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 910–927, 2006. Member-Wolfgang Emmerich.

[104] A. Jones and D. Cornford, "Defining the fluid framework," in *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration* (W. Chang and J. B. D. Joshi, eds.), pp. 695–700, 2007.

[105] Z. Reljic, "Extending geomatics concepts and capabilities for scientific visualization and communication: Integrating photorealism with geovisualization," Master's thesis, Department of Geography, University of Ottawa, November 2006.

[106] C. Claramunt, B. Jiang, and A. Bargiela, "A new framework for the integration, analysis and visualisation of urban traffic data within geographic information systems," *Transportation Research Part C: Emerging Technologies*, vol. 8, pp. 167–184, 2000.

[107] A. Jones and D. Cornford, "Advanced data driven visualisation for geo-spatial data," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 586–592, Springer, May 2006.

[108] A. Jones and D. Cornford, "A flexible, extensible object oriented real-time near photoreaslistic visualisation system: The system framework design," in *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling* (A. Riedl, W. Kainz, and G. Elmes, eds.), pp. 563–579, Springer-Verlag, 2006.

[109] K. Wilson, "Game object structure roundtable," in *Game Developers Conference Proceedings*, 2003.

[110] W. Buchanan, "A generic component library," in *Game Programming Gems 5* (K. Pallister, ed.), ch. 1.16, pp. 177–187, Charles River Media, February 2005.

[111] K. Ostermann and M. Mezini, "Implementing reusable collaborations with delegation layers," in *Object Oriented Programming, Systems, Languages, and Applications*, (Tampa Bay, FL, USA), 2001.

[112] Y. Smaragdakis and D. S. Batory, "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs," *Software Engineering and Methodology*, vol. 11, no. 2, pp. 215–255, 2002.

[113] U. P. Schultz, "Black-box program specialization," in *Proceedings of the 4th International Workshop on Component Oriented Programming*, p. 187, 1999.

[114] T. Weis, "Component customization," in *Proceedings of the 6th International Workshop on Component Oriented Programming*, (Budapest, Hungary), June 2001.

[115] B. Kücük, N. Alpdemir, and R. Zobel, "Customisable adapters for black-box components," in *Proceedings of the 3rd International Workshop on Component Oriented Programming*, 1998.

[116] M. Büchi and W. Weck, "Generic wrappers," in *Lecture Notes in Computer Science* (E. Bertino, ed.), pp. 201–225, 2000.

[117] G. Bobeff and J. Noye, "Molding components using program specialization techniques," in *Proceedings of the 8th International Workshop on Component Oriented Programming* (J. Bosch, C. Szyperski, and W. Weck, eds.), (Darmstadt, Germany), 2003. In conjunction with ECOOP 2003.

[118] Y. H. Mirza, "A compositional component collections framework," in *Proceedings of the 7th International Workshop on Component Oriented Programming*, (Malaga, Spain), 2002.

[119] D. H. Lorenz and J. Vlissides, "Designing components versus objects: a transformational approach," in *Proceedings of the 23rd International Conference on Software Engineering*, (Washington, DC, USA), pp. 253–262, IEEE Computer Society, 2001.

[120] D. Hamlet, "Component synthesis theory: The problem of scale," in *Proceedings of the 4th International Conference on Software Engineering Workshop on Component-based Software Engineering* (K. Wallnau, ed.), (Toronto, Canada), pp. 75–80, 2001.

[121] R. Reussner, "The use of parameterised contracts for architecting systems with software components," in *Proceedings of the 6th International Workshop on Component Oriented Programming* (W. Weck, J. Bosch, and C. Szyperski, eds.), June 2001.

[122] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[123] S. McDirmid, M. Flatt, and W. Hsieh, "Jiazzi: New-age components for old-fashioned Java," in *Object Oriented Programming, Systems, Languages, and Applications*, October 2001.

[124] S. McDirmid, M. Flatt, and W. Hsieh, "Mixing COP and OOP," in *Object Oriented Programming, Systems, Languages, and Applications* (D. H. Lorenz and V. C. Sreedhar, eds.), pp. 29–32, Technical Report NU-CSS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115, October 2001.

[125] J. Noble, "Three features for component frameworks," in *Proceedings of the 4th International Workshop on Component Oriented Programming*, p. 187, 1999.

[126] M. Chaudron, "Reflections on the anatomy of software composition mechanisms," in *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Vienna, Austria), September 2001.

[127] D. Birngruber, "A software composition language and its implementation," *Lecture Notes in Computer Science*, vol. 2244, pp. 519–529, 2001.

[128] R. Ierusalimschy, *Programming in Lua.* Lua.org, 2 ed., 2006.

[129] K. Henney, "Valued conversions," *From Mechanism to Method, C++ Report*, vol. 12, pp. 37–40, JulyAugust 2000.

[130] D. D. Corkill, "Blackboard systems," *AI Expert*, vol. 6, pp. 40–47, Sept. 1991.

[131] N. Llopis, "The beauty of weak references and null objects," in *Game Programming Gems 4* (A. Kirmse, ed.), ch. 1.7, pp. 61–68, Charles River Media, March 2004.

[132] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[133] G. R. Madey, G. Szabó, and A.-L. Barabási, "WIPER: The integrated wireless phone based emergency response system," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 417–424, Springer, 2006.

[134] D. Helbing, I. Farkas, and T. Vicsek, "Simulating dynamical features of escape panic," *Nature*, vol. 407, pp. 487–490, 2000.

[135] B. Plale, D. Gannon, D. Reed, S. Graves, K. Droegemeier, B. Wilhelmson, and M. Ramamurthy, "Towards dynamically adaptive weather analysis and forecasting in LEAD," in *International Conference on Computational Science* (V. S. et al., ed.), vol. 3515 of *Lecture Notes in Computer Science*, pp. 624–631, Springer, 2005.

[136] K. Mahinthakumar, G. von Laszewski, R. Ranjithan, D. Brill, J. Uber, K. Harrison, S. Sreepathi, and E. Zechman, "An adaptive cyberinfrastructure for threat management in urban water distribution systems," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 401–408, Springer, 2006.

[137] M. Parashar, V. Matossian, H. Klie, S. G. Thomas, M. F. Wheeler, T. Kurc, J. Saltz, and R. Versteeg, "Towards dynamic data-driven mangement of the Ruby Gulch waste repository," in *International Conference on Computational Science* (V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3993 of *Lecture Notes in Computer Science*, pp. 384–392, Springer, 2006.

[138] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. H. Rose, M. Rys, J. Siméon, and P. Wadler, "XQuery 1.0 and XPath 2.0 formal semantics." World Wide Web Consortium, Recommendation REC-xquery-semantics-20070123, January 2007.

[139] R. Taylor *et al.*, "A component- and message-based architectural style for GUI software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[140] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, pp. 717–734, September 1995.

[141] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the 5th European Software Engineering Conference*, (London, UK), pp. 137–153, Springer-Verlag, 1995.

[142] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems* (G. T. Leavens and M. Sitaraman, eds.), ch. 3, pp. 47–67, New York, NY: Cambridge University Press, 2000.

Chapter 6

Appendices

Appendix A

Publications

The following publications accompany this thesis.

Jones, A. and Cornford, D. Riedl, A.; Kainz, W. and Elmes, G. (ed.) A Flexible, Extensible Object Oriented Real-time Near Photoreaslistic Visualisation System: The System Framework Design Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling, Springer-Verlag, 2006, 563-579

Jones, A. and Cornford, D. Alexandrov, V.N.; van Albada, G.D.; Sloot, P.M. and Dongarra, J. (ed.) Advanced Data Driven Visualisation for Geo-spatial Data Computational Science - ICCS 2006, Springer, 2006, 3993, 586-592

Jones, A.; Mantle, C. and Cornford, D. Lim, I.S. and Duce, D. (ed.) Data Driven Graphical Applications: A Fluid Approach Theory and Practice of Computer Graphics Eurographics UK Chapter Proceedings, Eurographics Association, 2007, 187-194

Jones, A. and Cornford, D. Chang, W. and Joshi, J.B.D. (ed.) Defining the Fluid Framework Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration, 2007, 695-700